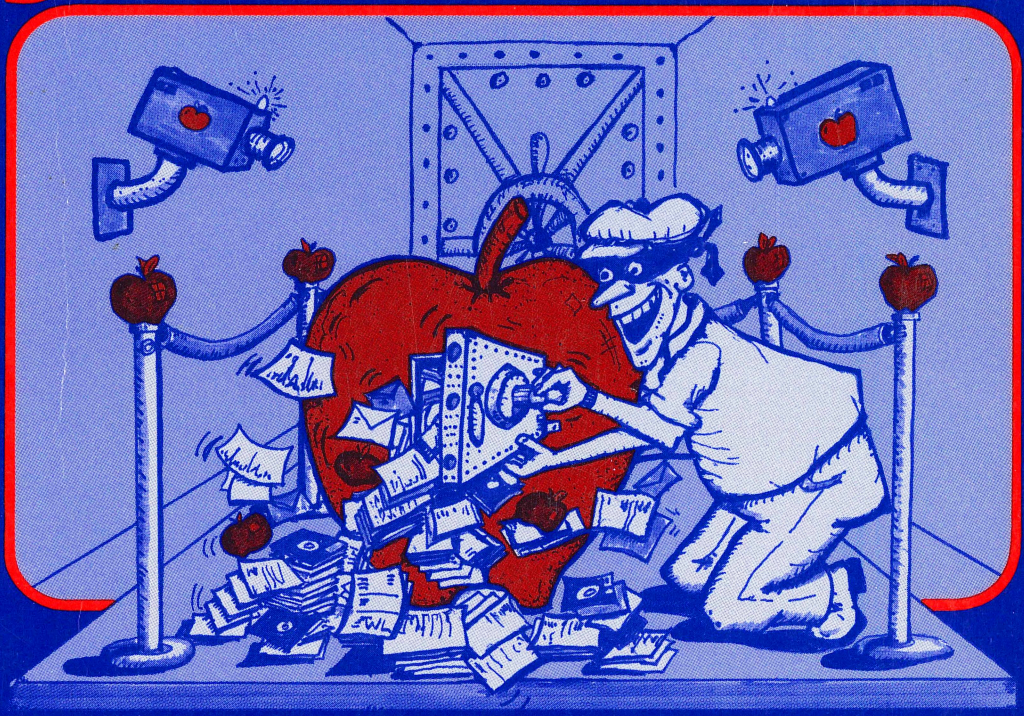
Apple Programming Secrets "THEY" didn't want you to know



Wayne Dyrness

P-H
PRENTICE–HALL PERSONAL COMPUTING SERIES

Apple Programming Secrets "They" Didn't Want You To Know

Prentice-Hall Personal Computing Series
Lance A. Leventhal, series editor

Anderson/Anderson, *The UNIX™ C Shell Field Guide*
Busch, *Sorry About the Explosion: A Humorous Guide to Computers*
Dickey, *Kids Travel on Commodore 64*
Fabbri, *Animation, Games, and Graphics for the Timex-1000*
Fabbri, *Animation, Games, and Sound for the Apple II/IIe*
Fabbri, *Animation, Games, and Sound for the Commodore 64*
Fabbri, *Animation, Games, and Sound for the IBM PC*
Fabbri, *Animation, Games, and Sound for the TI 99/4A*
Fabbri, *Animation, Games, and Sound for the VIC 20*
Glazer, *Managing Money with Your Commodore 64*
Glazer, *Managing Money with Your IBM PC*
Glazer, *Managing Money with Your VIC 20*
Harris & Scofield, *IBM PC Conversion Handbook of BASIC*
Lima, *dBASE II for Beginners*
Lima, *Mastering dBASE III in Less Than a Day*
McLaughlin & Boulding, *Financial Management with Lotus® 1-2-3®*
Nitz, *Business Analysis and Graphics with Lotus® 1-2-3®*
Scanlon, *EasyWriter II System Made Easy-er*
Scanlon, *Microsoft Word for the IBM PC*
Scanlon, *The IBM PC Made Easy*
Schnapp, *Macintosh Graphics in Modula-2*
Schnapp & Stafford, *Commodore 64 Computer Graphics Toolbox*
Schnapp & Stafford, *Computer Graphics for the Timex 1000 and Sinclair ZX-81*
Schnapp & Stafford, *VIC 20 Computer Graphics Toolbox*
Steinhauer, *Artificial Behavior: Computer Simulation of Psychological Processes*
Thro, *Making Friends with Apple Writer II*

Apple Programming Secrets "They" Didn't Want You To Know

Wayne Dyrness

PRENTICE-HALL, Englewood Cliffs, NJ 07632

Library of Congress Catalog Card Number: 85-63099.

Editorial/production supervision: Sophia Papanikolaou
Interior design: Sophia Papanikolaou
Cover design: Lundgren Graphics, Ltd.
Manufacturing buyer: Gordon Osbourne

© 1986 by Prentice-Hall
A Division of Simon & Schuster, Inc.
Englewood Cliffs, New Jersey 07632

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY:
The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-039215-4 025

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Whitehall Books Limited, *Wellington, New Zealand*

DEDICATION

This book is dedicated to my father, who taught me what it means to work hard; to my mother, whose love and support has been neverending; and
to my wife, whose patience and love were clearly evident throughout the writing of this book.

CONTENTS

PREFACE **xvii**

INTRODUCTORY CHAPTER **xix**

Goals of This Book *XX*
Criteria for Evaluating Comprehension *XX*

CHAPTER 1 IT'S MUSIC TO MY EARS **1**

Introduction (in the right key for music) *1*
Simple Sound Effects and Routines *2*
Programming the Notes into Your Apple *3*
Chart of Notes and Values *7*
Tone Tester *8*
Random Tone Generator *8*

Two-Octave Scale 9
The Eight-Note Piano 9
Another Way to Program Music 10
Examples of Music 11
Computer Version of "Name That Tune" 14
Special Sound Effects in APPLESOFT BASIC and INTEGER BASIC 20

CHAPTER 2 IT'S WHAT'S INSIDE THAT COUNTS! PEEKS, POKES, AND CALLS **26**

Introduction 26
Chart of PEEKs 29
Chart of POKEs 30
Chart of CALLs and WAITs 32

CHAPTER 3 THE TEXT WINDOW MASSACRE **34**

Dazzling Screen Displays 34
Left Margin 35
Width 35
Top Margin 37
Bottom Margin 37
Twenty-Four Lines of Information???! 000
Controlling the Cursor 38
SPC 41
Remembering Where the Cursor Is (PEEK and POKE 36 and 37) 41
Setting Limits on Inputs 45
PEEK (37) and POKE (37) (aexpr) 47
Home Is Where I Belong (CALL -936) 49
Watch Out Below! (CALL -958) 49
All Right! Out of Sight! (CALL -868) 49

CHAPTER 4 CLEAN SCREENS FOR THE CINEMATIC FANATIC **51**

Introduction 51
Erase a Line of Information 53
Erase Portion of Screen Only 53

Erase Entire Screen 53
"The Invisible Line" 53
"Bottoms Up!" 54
"The Blackboard Eraser" 54
Special Effect Screen Clears 55
Split the Seams! (Erase Screen from Middle Out) 55
Pull That Curtain Aside! (Erase Screen from Right to Left) 56
A Variation of Right to Left 56
Mixed Doubles 58
Exit, Stage Right (Left to Right Wipe) 59
Pull It Over to the Corner, Will Ya'! (Diagonal Screen Erase) 59
Not That Corner, The Other One! (Diagonal Erase, Number 2) 61
Journey to the Center of the Screen (Outside In) 62
Close the Curtains! (Erase from Sides Inward) 64
Inverted T Erase 65
Close the Window (Erase from Top and Bottom to Middle) 66
The Camera Lens (Erasure Inside Out) 67
The Noisy Camera Lens (Erase Inside Out with Tone) 68
Scroll Up 69

CHAPTER 5 SCREENS ALIVE! 71

Introduction 71
Typewriter Sound 71
Writing a String Backwards 72
Simple Version 72
Typewriter 72
Noisy, Backwards Billboard 73
Title Wave 73
Flashy Routines 74
Special Routines 75
Underlining a Word or Phrase 76
Not So Flashy Routines 76

Moving a Word or Phrase Across the Screen 77
Moving a String Left to Right 77
Moving a String Right to Left 77
Moving a String Up and Down the Screen 78
Coming Together 78
Move with the Sound 80
I've Been Framed! 80
Inverse Box on Screen 80
"Framed Again!" 82
Hello Frame 83
Star Frame 83
The Bulletin Board: Horizontal Scrolling 84
May I Quote You? 85
Centering Words or Phrases 85
Title Maker 86
Getting the Straight Answer 89
Spin Master Menu 90
Spin Writing 91
Inverse Field Input Subroutine 92
? Extra Not Ignored 94
What's That You Say?? 95
Read the Keyboard: PEEK (-16384), 95
Could You Repeat That, Please? 96
PEEK (49152), 96
Error-Proof User Input 96
Menu Magic 97

CHAPTER 6 YOUR ERROR-ONEOUS ZONE IS SHOWING 99

Introduction 99
String Functions 100
LEN 100
VAL 100
STR$ 100
ASC 102
CHR$ 103
LEFT$ 104
RIGHT$ 105
MID$ 105

Programming-Related Errors *107*
"Beyond Limit" Errors *107*
"Incorrect Type" Errors *108*
Interrupting a Program *108*
Error Trap INPUT Routines *109*
TRACE *112*
SPEED *113*
DOS-Related Errors *115*
MON and NOMON *115*
Using ONERR GOTO Routines *116*
How to Handle Error Codes *118*
DOS Error Code 4—Write Protected, 118
DOS Error Code 5—End of Data Error, 119
DOS Error Code 6—File Not Found, 119
DOS Error Code 8—I/O Error, 120
DOS Error Code 9—Disk Full, 120
DOS Error Code 10—File Locked, 120
DOS Error Code 11—Syntax Error, 120
DOS Error Code 12—No Buffers Available, 121
Programming Error Code 254—Bad Response to an Input Statement, 121
Programming Error Code 255—CTRL C Interrupt Attempted, 121
DOS Error Code 56—RESET, 121
Hints *122*
Hint 1 *122*
Hint 2 *122*
DOS Error Codes (Listing) *123*
Program Error Codes (Listing) *123*
Other Error Messages *124*
Can't Continue *124*
Illegal Direct *124*

CHAPTER 7 WHAT'S THE ROUTINE? **125**

Introduction *125*
CATALOG Lister *125*
Ampersand Kill—CATALOG Utility *126*
Cursor Dash *126*
Applebetize *126*
Alphabetize *127*
Bubble Sort *127*
Shell–Metzner Sort *128*

Column Alignment 128
Simple Alignment 128
Dollar and Cents 1 129
Dollar and Cents 2 129
An Averaging Routine 130
Mortgage Calculation 131
Probability 131
ASCII Table 132
Paddles or Joystick Connected? 132
Paddle Tester 132
Undelete Lost Files 132
"Undelete" Program 1 133
"Undelete" Program 2 135
Delete Arrays 136
PEEK 0 136
Time Clock 137
Countdown Clock 137
POKE 33,33 138
CTRL-D Problem 139
Signal Bell 140
Security Procedures 141
POKE Protection 142
CTRL-C 142
CTRL Characters 142
CTRL Character Revealed 143
Ask Questions 143
The Invisible Security Code 144
MASTERMIND 145
Random Numbers 153
Handy Subroutines 154
Calendar 155

CHAPTER 8 INTRODUCTION TO RANDOM-ACCESS AND SEQUENTIAL TEXT FILES 162

Sequential and Random-Access Text Files 163
Modular Programming 163
Clarity 164
Error Checking 166
Variable Names 166
String Variables 167

MAXFILES *168*
Disk Format and Disk Commands *171*
Disk Format *171*
Difference between Sequential and Random-Access Files *172*
Disk Commands *172*
Disk Commands Used in Text Files *176*

CHAPTER 9 SEQUENTIAL TEXT FILES **177**

Introduction *177*
How to Create a Sequential Text File *178*
Step 1—Open *178*
Steps 2 and 3—Write and Print *179*
Step 4—Close *180*
? Extra Ignored *183*
Retrieving Information from a Sequential Text File *183*
Step 1—Open *184*
Steps 2 and 3—Read and Input *184*
Adding Information to a Sequential Text File *186*
How to Make Changes in a Sequential Text File *188*
DELETE Text Files *191*
Copying Text Files *192*
Other Ways to Copy Files *193*
Alphabetizing Text Files *195*
Sort Routine *196*
POSITION *197*
BYTE Control *201*
Writing a BYTE *203*

CHAPTER 10 RANDOM-ACCESS TEXT FILES **205**

Introduction *205*
How to Create a Random-Access Text File *207*
Step 1—Open *208*
Steps 2 and 3—Write and Print *208*
Step 4—Close *210*

Retrieving Information from a Random-Access Text File *213*
Step 1—Open *213*
Steps 2 and 3—Read and Input *214*
Step 4—Close *214*
How to Add Information to a Random-Access Text File *215*
More on RO *217*
How to Make Changes in a Random-Access Text File *218*
A Quick Review *220*
DELETE Random-Access Text Files *221*
Copying a Random-Access Text File *221*
Another Way to Copy *222*
Alphabetizing a Random-Access Text File *223*
Searching for Specific Data *223*
BYTE Control *226*
Padding a String *228*

CHAPTER 11 ONE FINAL WORD **231**

How to Get Free or Inexpensive Software *231*
A Source for Free Software *233*
Computer Users' Group *234*
Acknowledgment *234*

APPENDIX A ASCII CHARACTERS **236**

APPENDIX B FILE TYPE USE, LISTED BY DOS COMMAND **237**

APPENDIX C DOS COMMANDS (EXECUTION MODES) **238**

APPENDIX D DOS COMMANDS—CATEGORIES **239**

APPENDIX E FILE PARAMETERS 240

APPENDIX F RESERVED WORDS IN APPLESOFT 241

GLOSSARY 242

BIBLIOGRAPHY 245

INDEX 246

PREFACE

"Say . . . I wonder how the programmer did that??!!"

That's a question we all may ask ourselves at one time or another while watching someone else's program run. Somehow the programmer has used a clever and eye-catching technique that stimulates your interest.

Like most computer users, you have probably developed your programming skills by examining how other programs are designed. You see a clever subroutine or technique, find out how it works, and then try to incorporate it into your own programs.

Unfortunately, many programs may be difficult to understand, or worse yet, may be written in a programming language you aren't familiar with. Your question "how did the programmer do that?" remains unanswered . . . until now.

Apple Programming Secrets "They" Didn't Want You To Know has the answers. It is a compilation of programs, subroutines, and special tricks that show you how to use advanced programming techniques. Many of these tips are not found in any manuals. And, best of all, they are written in BASIC.

This book is intended to be used as a reference manual. Its design and format will make it easy for you to locate special techniques to incorporate into your

programs. Be very careful as you type the routines into your computer. Always double-check them for misspelled words, incorrect numbers, proper spacing, etc., before running your program. *One mistake can create several problems and in some cases may destroy your entire program.*

You may want to create a library disk for many of the routines. Later, you can include them in a program by using the MERGE function in the RENUMBER program on your System Master diskette.

This book can also be educational. You can examine routines and techniques to see how they work. You can try rewriting a routine to make it more efficient. Be creative. Think of ways you can utilize these routines to solve some of your programming problems.

This book contains programs and routines the author uses. They are interesting and exciting to watch as well as useful.

We hope you will find *Apple Programming Secrets "They" Didn't Want You To Know* too good a secret to keep quiet about!

INTRODUCTORY CHAPTER

Have you ever waded through a computer book section of a bookstore searching for that one special book that will be worth your hard-earned money? Maybe that's what you're doing right now. The number of books is overwhelming.

"Which of all these books will help me learn more about programming?" you ask yourself.

I asked the same question. And after spending hours looking through shelves and shelves of books, I discovered that most computer programming books fall into two categories:

1. Highly technical books (e.g., on machine or assembly language), and
2. BASIC programming guides.

I was interested in the second category. BASIC is the most common language for personal computers. It is written in English and is easy to learn. "BASIC is the language I feel comfortable with," I thought.

I quickly discovered that almost all BASIC books seemed to be clones of each other. Every book presented the fundamentals of BASIC programming.

They described PRINT statements, FOR . . . NEXT loops, GOTOs, etc. These were things I already knew! After all, they are among the first things a beginning programmer learns.

There didn't seem to be any middle ground between learning the fundamentals of BASIC and programming in more advanced languages. Thus, the genesis of this book.

This book will benefit even the experienced programmer. Many programming skills covered very briefly in Apple manuals, or related material, are covered in greater depth here.

This book can easily be used in an educational environment. Much effort was taken to explain techniques clearly and simply. Humorous cartoons and fun-to-watch programs keep the interest level high.

GOALS OF THIS BOOK

The book can help you learn how to

1. Write practical programs that are also fun.
2. Learn to use PEEKs, POKEs, and CALLs to control the screen display.
3. Integrate the programming tricks and techniques you learn here into other programs.
4. Design your own special programs.
5. Make the computer work for you.
6. Correct your programming errors.
7. Create and use sequential and random-access text files.

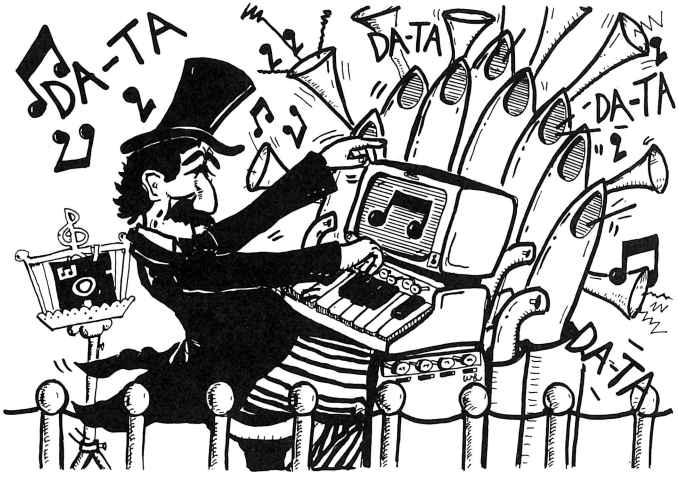
I hope the things you discover in this book will help you have fun while learning.

CRITERIA FOR EVALUATING COMPREHENSION

A teacher could use one or more of the following techniques to develop chapter tests.

1. Require an original short program utilizing at least two concepts from the chapter.
2. Create a series of true-or-false questions covering key points.

3. Require a routine similar to one presented, for example, a different type of screen clear.
4. Have the students write a program that demonstrates an understanding of POKE 36,CH; for example, a program that uses different margins and input techniques.
5. Ask the student to successfully trap at least five errors in a program using ONERR GOTO.



1 It's Music to My Ears

INTRODUCTION (in the right key for music)

"All my computer does is beep when I press the wrong key."

Sooner or later, that irritating beep leads you to ask,

"If the computer can generate a tone when I make a mistake, can it also generate other tones? Possibly music??!!"

Well, hold onto your hat! Your Apple can make music!

In fact, your Apple is very much like a musical instrument. It can play an infinite number of songs if you know how to "play" it.

The Apple computer is *monophonic* (one sound). Other monophonic instruments such as the clarinet, trumpet, or saxaphone are able to play only one note at a time as well. The piano, on the other hand, which is *polyphonic* (many sounds), is able to produce several notes together, as in chords.

Some computers can generate polyphonic chords by built-in sound generators.

Unfortunately, the Apple has no real sound generators except for the "clicking of the speaker."

When you make a mistake typing a program, the computer lets you know by "clicking the speaker" for a certain amount of time at a specific frequency, thereby producing a tone.

If we could control the pitch and duration of a tone and send it out of the Apple's built-in speaker, we could then create music by combining several different tones.

Well, you *can* do this! Now the question is no longer "if," but "how?"

"How can I make music on my Apple?"

This chapter will help you become a master Apple virtuoso. Armed with the **secrets** you're about to learn, you can turn any dull, simple program into an exciting, dynamic, attention-grabbing **sound** experience!

SIMPLE SOUND EFFECTS AND ROUTINES

There are two ways to access the speaker to produce tones: from BASIC or from machine language. Since this book does not cover machine language programming, we shall consider BASIC only.

In BASIC, **PEEK (−16336)** helps us produce our first sound. Typing

```
10  S = PEEK (-16336)

RUN
```

produces a click from the speaker. Albeit a humble beginning, it is our first step toward making music.

The number −16336 is a memory location in the computer. PEEKing into it produces a click from the Apple's built-in speaker. When we combine several PEEKs at that location, we get a series of clicks or a buzz.

We can create a buzz by putting this instruction within a FOR NEXT loop:

```
10  FOR J = 1 TO 50: S = PEEK (-16336): NEXT J
```

We can change the tone or pitch of the buzz by changing the number of PEEKs in a statement; i.e.,

```
10  FOR J = 1 TO 50
20  S = PEEK (-16336)
30  BUZZ = PEEK (S) + PEEK (S)
40  NEXT J
```

Try different combinations by adding or subtracting PEEK(S).

You can use tones like these within your programs in many ways:

1. In games when an object hits another object.
2. In error routines to call attention to a mistake.
3. In instructions with FLASH and bells to draw attention to *IMPORTANT WARNINGS*.
4. The list could go on and on, but I think you get the idea.

Chapter 5 discusses other ways to use PEEK (−16336), i.e., simulation of a typewriter, moving a phrase across the screen, etc.

Combining sound with pictures always stimulates the audiences interest. Just be careful that you don't overuse it. As exciting as buzzes, beeps, and dings are, they still "ain't playing my tune."

PROGRAMMING THE NOTES INTO YOUR APPLE

To control a tone's pitch and duration, we need a ***tone routine.*** We can create one by POKEing values into certain memory locations in the computer. We can then use a CALL routine to produce the desired tone.

The following is the skeleton program to POKE these values in.

Create a Tone Generator

```
 10 REM TONE ROUTINE
 20 :
    :
    :
100 GOSUB 3010: REM LOAD TONE GENERATOR
120 :
130 :
    :
    :
500 END
3000 REM TONE GENERATOR
3010 FOR J = 770 TO 790: READ D: POKE J,D: NEXT J: RESTORE:
     RETURN
3040 DATA 173,48,192,136,208,5,206,1,3,240,9,202,208,
          245,174,0,3,76,2,3,96
```

LISTING 1

In line 3010, we POKEd the tone generator into memory locations 770 through 790. The DATA line in 3040 gives the values.

In a sense, we have created the instrument. A piano is capable of producing tones, but will not do so until someone plays it. Similarly, your computer can now produce tones, but won't until you tell it what tones you want to hear. Adding line 120 to our new program gives us the "key" (pun intended) to producing music:

```
120  POKE 768,P: POKE 769,D: CALL 770
```

P will define the pitch and **D** will define the duration of the pitch (how long you want to hear the tone).

Pitch (**P**) must be a number between 0 and 255. The *higher* the number, the *lower* the tone produced. For example, a pitch with the value 255 sounds lower than a pitch with the value of 128.

Duration (**D**) must also be a number between 0 and 255. The larger the number, the longer the tone will be held before the next tone is played.

Let's try producing the pitch "C" (**P** value = 96) and hold it as a whole note (**D** value = 240). Change line 120 to

```
120  POKE 768,96: POKE 769,240: CALL 770
```

It sounds much more musical than the buzzes we produced earlier. We can produce several different tones of varying pitch and duration and combine them to produce music. However, it would soon become tedious and time-consuming to repeat the same line (such as 120) with different **P** and **D** values. Fortunately, we can rewrite line 120 as a subroutine as follows:

```
  20  REM TONE ROUTINE PROGRAM

 100  GOSUB 3010: REM LOAD TONE GENERATOR
 120  P = 96: D = 240: GOSUB 2000: REM PRODUCE TONE
 140  P = 85: D = 240: GOSUB 2000
 160  P = 76: D = 240: GOSUB 2000
 180  P = 85: D = 240: GOSUB 2000
 200  P = 96: D = 240: GOSUB 2000
 500  END
2000  REM PLAYS TONE
2010  POKE 768,P: POKE 769,D: CALL 770
2020  RETURN

3000  REM TONE GENERATOR
3010  FOR J = 770 TO 790: READ D: POKE J,D: NEXT J:RESTORE
      : RETURN
3040  DATA 173,48,192,136,208,5,206,1,3,240,9,202,208,
           245,174,0,3,76,2,3,96
```

LISTING 2

Note that our old line 120 is now a subroutine in lines 2000–2020. This makes it very easy to create music anywhere in a program by simply defining the variables **P** and **D** and sending the program to this subroutine (i.e., GOSUB 2000).

Although this technique is easier than what we did in our old line 120, it still has limitations. It's fine for short four-to-five-note passages, but if you want to create a tune of any length, you'll find this gets tedious as well.

The best way to store values of **P** and **D** is in DATA lines. Within a FOR NEXT loop, these values can be read and played very quickly.

The following is a program that demonstrates this.

```
 10 GOSUB 200: REM LOAD TONE GENERATOR
 20 FOR X = 1 TO 8
 30 READ P,D
 60 POKE 768,P: POKE 769,D: CALL 770
 70 NEXT X
 80 END
190 REM TONE GENERATOR
200 DATA 173,48,192,136,208,5,206,1,3,240,9,202,208,
        245,174,0,3,76,2,3,96
220 FOR J = 770 TO 790: READ D: POKE J,D: NEXT J
230 RETURN
600 REM PITCH AND DURATION VALUES
610 DATA 255,60,203,60,172,60,203,60,255,60,128,60,
        128,60,255,240
```

LISTING 3

Line 30 reads the pitch and duration from the DATA in line 610. Line 60 plays the note. Line 70 sends the program back in the FOR NEXT loop to enter another note. And there you have it. Music made easy!

A programmer may locate the machine language tone generator beginning at some address other than 770. This might be done to avoid writing data in the same memory locations used by other parts of the main program. In listing 4a, a program begins at memory location 771. Naturally, the routine is called by CALL 771 rather than CALL 770, as in Listing 3. Also note that the POKE locations in line 60 and the DATA values on line 200 are different. The notes' pitch and duration values (line 610) remain the same.

```
 10 GOSUB 200: REM LOAD TONE GENERATOR
 20 FOR X = 1 TO 8
 30 READ P,D
 60 POKE 0,P: POKE 1,D: CALL 771
 70 NEXT X
 80 END
```

```
190 REM TONE GENERATOR
200 DATA 173,48,192,136,208,5,206,1,3,240,9,202,208,
        245,174,0,3,76,2,3,96
220 FOR J = 770 TO 790: READ D: POKE J,D: NEXT J
230 RETURN
600 REM PITCH AND DURATION VALUES
610 DATA 255,60,203,60,172,60,203,60,255,60,128,60,
        128,60,255,240
```

LISTING 4a

Listing 4b is a tone program in a different memory location. The program is self-documented.

```
100 DATA 173,48,192,136,208,5,206,173,3,240,9,202,208,245,174,
        172,3,76,174,3,96
200 FOR J = 1 TO 21: READ X: POKE 941 + J,X: NEXT J
300 TEXT : HOME : VTAB 5: PRINT "INSTRUCTIONS FOR APPLESOFT
    TONE WITH DOS"
400 PRINT TAB( 10)"BY [ DOGFISH SOFTWORKS ]"
500 VTAB 10: PRINT "TO USE THIS TONE ROUTINE FIRST POKE AS
    FOLLOWS:":
600 PRINT "THE FOLLOWING LIST OF DATA:": PRINT : PRINT "(TOUCH
    A KEY TO CONTINUE)": CALL - 756
700 HOME : LIST 100,200
800 PRINT : PRINT "THIS WILL PLACE THE TONE ROUTINE IN": PRINT
    "MEMORY STARTING AT 942 ($3AE) TO": PRINT "963 ($3C2)."
900 PRINT : PRINT "TO USE THIS, USE THE SAME TYPE OF POKES":PRINT
    "AS YOU WOULD WITH THE ROUTINE IN THE":PRINT "APPLE II REFERENCE
    MANUAL WITH THE"
1000 PRINT "FOLLOWING DIFFERENCES:": PRINT
1100 PRINT "INSTEAD OF 0 FOR FREQUENCY, POKE 940": PRINT "INSTEAD
    OF 1 FOR DURATION, POKE 941"
1200 PRINT "INSTEAD OF CALL 2, USE CALL 942": PRINT
1300 PRINT "(TOUCH A KEY TO CONTINUE)": CALL - 756: HOME
1400 VTAB 3: PRINT "I WILL NOW BEGIN THE FOLLOWING TONE TEST "
1500 LIST 1600,1700
1600 FOR J = 1 TO 255: POKE 940,J: POKE 941,10: CALL 942
1700 NEXT I: END
```

LISTING 4b

Source: Public Domain Software

CHART OF NOTES AND VALUES

Your Apple has a musical range of more than four octaves. Most people have only a two-octave singing range. Tables 1-1 and 1-2 give the pitch and duration POKE values.

Also included is Table 1-3[1] of rest values. For those of you who are not musicians, rests are pauses before or after notes that are used in music to add effect. "Rest" assured that if you don't know how to read or write music, there are several examples of tunes and other short tone sequences at the end of the chapter.

TABLE 1-1 PITCH (P)

G = 255	G' = 128	G'' = 64	G''' = 31	G'''' = 15
Ab = 243	Ab' = 121	AB'' = 60	AB''' = 29	Ab'''' = 14
A = 231	A' = 114	A'' = 56	A''' = 28	A'''' = 13
Bb = 217	Bb' = 108	Bb'' = 53	Bb''' = 26	Bb'''' = 12
B = 203	B' = 102	B'' = 50	B''' = 25	
C = 192	C' = 96	C'' = 47	C''' = 23	
C# = 182	C#' = 90	C#'' = 45	C#''' = 22	
D = 172	D' = 85	D'' = 42	D''' = 21	
Eb = 162	Eb' = 80	Eb'' = 40	Eb''' = 20	
E = 154	E' = 76	E'' = 37	E''' = 18	E'''' = 11
F = 146	F' = 72	F'' = 35	F''' = 17	F'''' = 10
F# = 137	F#' = 67	F#'' = 33	F#''' = 16	F#'''' = 9

TABLE 1-2 DURATION (D)

Whole note = 240	Sixteenth note = 15
Half note = 120	Thirty-second note = 8
Quarter note = 60	Sixty-fourth note = 4
Eighth note = 30	One hundred and twenty-eighth note = 2

TABLE 1-3 RESTS (R)

Line # FOR R = 1 RV (rest value):NEXT R

Whole rest = 500
Half rest = 250
Quarter rest = 125
Eighth rest = 62
16th rest = 31
32nd rest = 16
64th rest = 8
128th rest = 4

[1]Published InCider Magazine, Wayne Green Inc., May 1983.

TONE TESTER

Below is a tone tester. It lets you enter a frequency and duration value. Then it plays the note value for the duration indicated. Type the program and experiment with it.

```
 10 REM TONE TESTER
 20 :
 30 HOME : VTAB 12
 40 FOR X = 770 TO 790: READ B: POKE X,B: NEXT X
 50 DATA 173,48,192,136,208,5,206,1,3,240,9,202,
        208,245,174,0,3,76,2,3,96
 60 INPUT "FREQUENCY (1 TO 255) ";F
 70 INPUT "DURATION (1 TO 255) ";D
 80 POKE 768,F: POKE 769,D: CALL 770
 90 PRINT : PRINT "ANOTHER TONE (Y/N)? ";
100 GET A$: PRINT A$: IF A$ = "Y" THEN 60
110 END
```

RANDOM TONE GENERATOR

Often, you may want to use the random tone generator (**RTG**). "What is that?" you ask. Simply a tone generator which plays random notes.

Many computer games use the RTG. The computer version of "Mastermind" in Chapter 7 demonstrates the RTG; random notes give the illusion that the computer is thinking of colors for you to guess and match. Actually, the computer needs very little time to come up with colors. The RTG is used strictly for effect.

Other game applications include balls bouncing, collisions, etc.

```
 10 REM RANDOM TONE GENERATOR
 20 GOSUB 200: REM LOAD TONE GENERATOR
 30 FOR X = 1 TO 10: REM WILL CREATE 10 RANDOM NOTES
 40 S = INT ( RND (1) * 200 ) + 10
 60 POKE 0,S: POKE 1,10: CALL 771
 70 NEXT X
 80 END

200 DATA 173,48,192,136,208,4,198,1,240,8,202,208,
        246,166,0,76,3,3,96
220 FOR J = 771 TO 789: READ X: POKE J,X: NEXT J
230 RETURN
```

LISTING 5

You can change how many notes will be generated by changing the value in the FOR NEXT loop in line 30, i.e., FOR X = 1 TO (number of notes you want). The duration of each note may be lengthened or shortened by changing the number POKEd at memory location 1 (line 60) to any number between 1 and 255. See Chapter 4, "The Noisy Camera Lens," for another example of how the RTG is used.

TWO-OCTAVE SCALE

On a scale (pun intended) from 1 to 8, how would you rate this tune?

```
 10 GOSUB 210: REM LOAD TONE GENERATOR
 20 FOR X = 1 TO 15
 30 READ P
 40 POKE 768,P: POKE 769,200: CALL 770
 50 NEXT X
 60 END

200 REM TONE GENERATOR
210 DATA 173,48,192,136,208,5,206,1,3,240,9,202,208,
        245,174,0,3,76,2,3,96
220 FOR J = 770 TO 790: READ D: POKE J,D: NEXT J
230 RETURN
600 REM NOTE VALUES
610 DATA 255,231,203,192,172,154,137,128,114,102,
        96,85,76,67,64
```

LISTING 6

THE EIGHT-NOTE PIANO

We have seen how to create music in a program. But can we make a specific key produce a specific note like a piano or organ does?

The following program shows how:

```
10 GOSUB 200: REM LOAD TONE GENERATOR
20 FOR X = 1 TO 8
30 READ N(X): REM READ NOTES
40 NEXT X
60 HOME
70 D = 50: REM BEGINNING DURATION
80 PRINT "TO PLAY THE PIANO. PRESS KEYS 1-8."
90 PRINT "TO CHANGE DURATION. PRESS ANY OTHER KEY."
```

```
100 K = PEEK (49152): IF K < 160 THEN 100
110 POKE 49168;0: IF K < 177 OR K > 184 THEN D = D
    + 25 - 250 * (D = 250): VTAB 8:PRINT "DURATION: ";D;" ":
    GOTO 100
120 K = K - 176: VTAB 7: PRINT "PITCH: ";K:
    POKE 0,N(K): POKE 1,D: CALL 771: GOTO 100

200 DATA 173,48,192,136,208,4,198,1,240,8,202,208,
    246,166,0,76,3,3,96
220 FOR I = 771 TO 789: READ D: POKE I,D: NEXT I
230 RETURN
600 REM NOTE VALUES
610 DATA 96,85,76,72,64,56,50,47
```

LISTING 7

Line 10 loads the tone generator. Lines 20 through 40 load the note values. Line 70 sets a default duration value. Line 100 reads the keyboard for any key pressed. Line 110 clears the keyboard scan. If any key other than 1 through 8 is pressed, the duration value changes; then the program prints to the screen and returns to wait for the next key to be pressed. Line 120 determines which key (1 through 8) has been pressed, plays it, prints it to the screen, and returns to wait for the next key press. For further information on PEEK 49152, see Chapter 5.

ANOTHER WAY TO PROGRAM MUSIC

Another way to produce music is to use the ampersand (&). The chart of POKEs in Chapter 2 gives several ways to use the ampersand. The & command is also called the *ampersand vector* or *and hook*.

Pressing any key and RETURN, other than numbers or the colon (:), produces a ?Syntax Error message. However, if you press & and RETURN, the computer will accept it without error.

The reason is that whenever ". . . the Apple interpreter finds an & in a line of BASIC, it jumps to a certain address in memory. That address normally contains an instruction that causes a return to the calling program, so nothing seems to happen, but that only means that the & hook is not connected to anything. You can change that and connect the & hook to a routine of your own."[2]

There are three memory locations that hold instructions regarding the ampersand hook. Their addresses are 1013, 1014, and 1015.

[2]Jack Root, "Everyone's Guide to Assembly Language," *Softalk,* November 4, 1983, p. 119.

You may have several programs on a diskette which use music routines. Rather than rewriting the tone generator for each program, you can create a short **POKE FOR MUSIC** program, as shown in Listing 4a.

Once you've RUN it, the tone generator remains in memory, unless, of course, some other program wants to use the same memory locations and overwrites it. You might include it in the HELLO program of your game diskette so that booting the disk installs the tone generator.

In the following short program **POKE FOR MUSIC,** we are setting up the ampersand hook (line 50) to act like the POKE and CALL routine we used in Listing 7 to produce music. You may notice that the starting address of the memory location and the DATA information are different from those in Listing 7.

```
10  FOR J = 768 TO 833: READ P: POKE J,P: NEXT J

20  DATA 201,84,208,15,32,177,0,32,248,230,138,72,32,
         183,0,201,44,240,3,76,201,222,32,177,0,32,248,230
30  DATA 104,134,3,134,1,133,0,170,160,1,132,2,173,48,
         192,136,208,4,198
40  DATA 1,240,7,202,208,246,166,0,208,239,165,3,133,
         1,198,2,208,241,96

50  POKE 1013,76: POKE 1014,0: POKE 1015,3
60  PRINT CHR$(4);"RUN (insert program name here)"
```

LISTING 8

See the next section for some examples of using the ampersand hook in music.

EXAMPLES OF MUSIC

Bourrée (an original from "moi")

This melody is from an original four-part bourrée that I wrote years ago. Many of the musical compositions from the baroque era were experimentations in four-part counterpoint. Johann Sebastian Bach was the most notable composer who utilized and made great advances in four-part writing. Often, there were no words to the songs. The singers would instead sing words like "do" or "do-bee do," etc.

Unfortunately, with a monophonic instrument such as our Apple, the counterpoint cannot be heard. However, as you listen to this tune, try to imagine how the other three parts would sound.

Note that this song utilizes the ampersand hook technique. Remember to RUN POKE FOR MUSIC (Listing 8) before RUNing this program.

```
2000 REM BOURRÉE
2010 FOR B = 1 TO 14
2020 READ X,Y
2040 & TX,Y
2050 NEXT B
2060 DATA 50,100,60,100,67,100,75,100,60,100,50,250
2070 DATA 50,200,60,100,57,100,57,50,60,50,67,100,57,
        100,60,255
2090 FOR B = 1 TO 32
2095 READ X,Y: & TX,Y: NEXT B
3000 DATA 60,50,67,50,75,200,67,100,90,250
3010 DATA 67,50,75,50,80,100,67,100,50,100,100,100,57,250
3020 DATA 90,50,80,50,75,100,100,250,57,100,60,250
3030 DATA 90,50,80,50,75,100,100,200,60,100,67,250,67,
        50,75,50,80,100
3040 DATA 67,100,90,100,67,100,100,250,100,250
3045 RESTORE: FOR B = 1 TO 87: READ X,Y: & TX,Y: NEXT B
3050 DATA 100,50,90,50,100,50,90,50,80,50,75,50
3060 DATA 80,50,75,50,67,50,60,50,67,50
3070 DATA 60,50,57,50,50,250
4000 FOR B = 1 TO 16
4010 READ X,Y: & TX,Y: NEXT B
4020 DATA 44,100,90,100,50,100,57,100
4025 DATA 67,50,60,50,67,100,100,200
4030 DATA 57,100,100,100,57,100,60,100
4035 DATA 75,50,67,50,60,100,100,200
4040 DATA 60,100,100,100,60,100,67,100
4045 DATA 80,50,75,50,67,100,100,250
4050 DATA 60,100,67,50,75,50,80,50
4060 DATA 90,50,100,100,67,100,50,100
4070 DATA 100,100,57,250,50,100
4075 DATA 90,50,80,50,75,50,100,50
4080 DATA 120,50,100,50,150,255,150,250
4090 END
```

LISTING 9

Here is another example of ampersand hook music:

Introduction Music

```
2000 REM SUB INTRO MUSIC
2010 PRINT "WELCOME TO THE EXCITING GAME OF"
2020 & T100,100: & T100,50: & T100,50: & T75,66: & T100,66:
```

```
      & T75,66: & T60,255
2030  PRINT: PRINT: PRINT "THAT'S RIGHT... YOU GUESSED IT..."
2040  FOR X = 1 TO 2000: NEXT X
2050  VTAB 12: HTAB 12: FLASH: PRINT "NAME THAT TUNE!!":NORMAL
2060  VATB 22: INPUT "DO YOU WANT INSTRUCTIONS ";Y$: IF LEFT $
      (Y$,1) < > "Y" THEN RETURN
2070  :
2080  REM INSTRUCTIONS
2090  :
2100  REM USE THE FOLLOWING LINES TO PRINT INSTRUCTIONS
2110  :
2120  :
2130  RETURN
```

LISTING 10

Close Encounters Theme

```
  10  REM CLOSE ENCOUNTERS THEME
  20  GOSUB 200: REM INSTALL TONE GENERATOR
  30  FOR X = 1 TO 5
  40  READ P,D
  50  POKE 768,P: POKE 769,D: CALL 770
  60  NEXT X
  70  END

 200  REM TONE GENERATOR

 200  DATA 173,48,192,136,208,5,206,1,3,240,9,202,208,
           245,174,0,3,76,2,3,96
 220  FOR J = 770 TO 790: READ D: POKE J,D: NEXT J
 230  RETURN
 600  REM NOTE VALUES
 610  DATA 114,240,102,240,128,240,255,240,172,255
```

LISTING 11

Charge!

```
  10  GOSUB 210: REM INSTALL TONE GENERATOR
  20  FOR X = 1 TO 17
  30  READ P,D
  40  POKE 0,P: POKE 1,D: CALL 771
  50  NEXT X
  60  END
```

```
200 REM TONE GENERATOR
210 DATA 173,48,192,136,208,4,198,1,240,8,202,208,
        246,166,0,76,3,3,96
220 FOR J = 771 TO 789: READ X: POKE J,X: NEXT J
230 RETURN

300 REM TONES CHARGE!
310 DATA 192,60,154,60,128,60,128,120,154,60,128,240
320 DATA 96,30,96,30,96,255
```

LISTING 12

Reveille

```
 10 GOSUB 210: REM INSTALL TONE GENERATOR
 20 FOR X = 1 TO 17
 30 READ P,D
 40 POKE 0,P: POKE 1,D: CALL 771
 50 NEXT X
 60 END

200 REM TONE GENERATOR
210 DATA 173,48,192,136,208,4,198,1,240,8,202,208
        246,166,0,76,3,3,96
220 FOR J = 771 TO 789: READ X: POKE J,X: NEXT J
230 RETURN

300 REM REVEILLE
310 DATA 255,60,192,60,154,30,192,30,255,60,
        154,60,192,60,154,30
320 DATA 192,30,255,60,154,60,192,60,154,30,192,30,
        255,60,255,60,192,255
```

LISTING 13

COMPUTER VERSION OF "NAME THAT TUNE"

This program is the computer version of that popular game show, "Name That Tune." There are two songs already built into the program in lines 1500 through 1560. To see this program at its best, add eight to ten more songs of your choosing.

Since the game will test only on songs in the program, you may want to add

more songs and store them in text files under SONG.1, SONG.2, SONG.3, and so on. Add some instructions near the beginning of your program to open these files at random. Then you would have no idea which songs will be played.

```
  5 REM NAME THAT TUNE!
 10 TEXT
 20 :
 40 NS = 2: REM NS=# OF SONGS IN YOUR PROGRAM
 50 SP = 1
 55 :
 60 GOSUB 1320: REM INSTALLS TONE GENERATOR
 65 :
 66 :
 67 REM INTRODUCTION
 68 :
 69 :
 70 HOME
 80 PRINT "LADIES AND GENTLEMEN"
 90 N = 17: REM # OF NOTES IN TUNE
100 GOSUB 1360: REM PLAYS TUNE
110 PRINT : PRINT "PRESENTING..."
120 N = 6: GOSUB 1360
130 HOME
140 HTAB 14: FLASH : PRINT "NAME THAT TUNE.": NORMAL
150 N = 3: GOSUB 1360
160 PRINT : PRINT "YES, LADIES AND GENTLEMEN! IT'S
    THAT ": PRINT "EXCITING GAME OF FUN AND KNOWLEDGE,
    OF": PRINT "SIGHT AND SOUND!"
170 PRINT : PRINT : HTAB 20: PRINT "IT'S"
180 FOR X = 1 TO 5000: NEXT X : REM TIME DELAY
190 HOME
200 :
210 :
220 REM CREATE WHIRRING SOUND
230 FOR X = 255 TO 12 STEP - 10: POKE 0,X: POKE 1,4:
    CALL 771: NEXT X
240 :
250 :
260 REM INSTRUCTIONS
270 :
280 :
290 HTAB 14: PRINT "NAME THAT TUNE"
300 :
310 PRINT : PRINT
320 PRINT "HOW MANY NOTES DO YOU NEED TO NAME THAT":
    PRINT "TUNE? YOU WILL BE ASKED THAT QUESTION"
330 PRINT "AND IT'S UP TO YOU TO ANSWER CORRECTLY."
340 PRINT:PRINT " IF YOU MAKE A WRONG GUESS, YOU'LL": PRINT
```

```
" BE GIVEN ONE MORE NOTE. YOU": PRINT " ONLY GET THREE
CHANCES OR YOU LOSE!"
350 PRINT:PRINT
360 PRINT "FOR EVERY WRONG GUESS YOU MAKE, YOU LOSE 5 POINTS."
370 PRINT:PRINT "IF YOU GUESS CORRECTLY ON THE FIRST": PRINT
"GUESS, YOU'LL BE AWARDED 15 POINTS!"
380 PRINT "YOUR 2ND GUESS IS WORTH 7 POINTS, AND YOUR THIRD IS
WORTH 5 POINTS."
390 VTAB 23: PRINT "PRESS ANY KEY TO CONTINUE.";: GET A$: PRINT
400 HOME
410 PRINT "IF YOU'RE READY FOR THE FIRST TUNE, PRESS":PRINT
"<RETURN>."
420 VTAB 23: GET A$: PRINT
430 :
440 :
450 REM START OF GAME
460 :
470 :
480 :
490 GOSUB 610: REM READS NOTE VALUES
495 HOME
500 PRINT "THIS SONG HAS ";TN;" NOTES.": PRINT
505 POKE 34,4: REM RESETS TOP MARGIN
506 HOME
510 GU = 1 : REM GU = GUESS
520 PRINT "HOW MANY NOTES DO YOU": PRINT "NEED TO ";: FLASH:
PRINT "NAME THAT TUNE?";: NORMAL: INPUT " ";NN$
522 CALL -958
525 IF NN$ = "" THEN CALL -198: VTAB PEEK(37) - 1: HTAB 1: GOTO 520
530 IF VAL (NN$) > TN THEN CALL -198: CV = PEEK (37) - 1: PRINT:
PRINT "TOO MANY NOTES ENTERED. TRY AGAIN.": VTAB CV: HTAB 1:
GOTO 520
540 IF VAL (NN$) > TN - 2 THEN CALL -198: CV = PEEK (37) -1: PRINT:
PRINT "YOU HAVE THREE CHANCES. THE MAXIMUM I'M": PRINT "GOING
TO GIVE YOU IS ";TN - 2;" TRIES. RE-ENTER. ": VTAB CV: HTAB 1:
GOTO 520
550 PRINT: PRINT
560 :
570 PRINT "O.K. HERE ARE YOUR ";NN$;" NOTES."
580 N = VAL (NN$)
585 :
590 GOSUB 670: REM PLAYS NOTES
595 :
600 GOTO 710
605 REM SUB TO READ NOTE VALUES
610 READ TN,NM$
620 DIM P(TN),D(TN)
630 FOR X = 1 TO TN
```

```
640 READ P(X),D(X)
650 NEXT X
660 RETURN
665 REM SUB TO PLAY NOTES
670 FOR X = 1 TO N
680 POKE 0,P(X): POKE 1,D(X): CALL 771
690 NEXT X
700 RETURN
710 PRINT
720 PRINT "GUESS NUMBER ";GU: PRINT
730 PRINT "WHAT IS THE NAME OF THE SONG?": INPUT
    "ENTER → ";GU$
735 :
740 IF GU$ < > NM$ THEN 890: REM COMPARE GUESS
    WITH CORRECT ANSWER
745 :
746 :
747 REM WINNER!!
748 :
749 :
750 FOR X = 1 TO 10: CALL - 198: NEXT X
754 TEXT
755 HOME
760 PRINT : PRINT : PRINT "YOU ARE CORRECT!!
770 IF GU < > 1 THEN GOTO 810
780 PRINT "YOU ARE AWARDED 15 POINTS"
790 SC = SC + 15: REM ADD TO SCORE
800 GOTO 990
810 IF GU < > 2 THEN GOTO 850
820 PRINT "YOU ARE AWARDED 7 POINTS."
830 SC = SC + 7: REM ADD TO SCORE
840 GOTO 990
850 PRINT "YOU ARE AWARDED 5 POINTS."
860 SC = SC + 5: REM ADD TO SCORE
870 GOTO 990
872 :
874 :
880 REM YOU LOSE!
882 :
884 :
890 FOR J = 1 TO 50:S = PEEK ( - 16336) + PEEK
    ( - 16336): NEXT
900 HOME
910 SB = 5: REM SUBTRACT 5 POINTS
920 PRINT : PRINT "BAD GUESS!!!"
930 PRINT : PRINT "YOU LOSE ";SB;" POINTS!":SC = SC - SB
940 PRINT "YOUR SCORE IS NOW: ";SC
950 IF GU = 3 THEN PRINT: PRINT "NO MORE GUESSES.": GOTO 982
```

```
 955 :
 960 GU = GU + 1: REM INCREASE GUESS NUMBER
 965 :
 970 PRINT "TRY AGAIN.": PRINT "HERE'S THE TUNE WITH ONE MORE NOTE.":
     PRINT: FOR X = 1 TO 2000: NEXT X
 975 :
 980 N = N + 1: GOSUB 670: GOTO 710
 981 :
 982 PRINT : PRINT "THE SONG IS: ";: FLASH: PRINT NM$: NORMAL
 983 N = TN: GOSUB 670: REM PLAY ENTIRE SONG
 984 TEXT: REM RESTORE TEXT WINDOW TO NORMAL
 985 :
 990 REM COMPARE SONGS PLAYED TO NUMBER OF SONGS IN THE PROGRAM
1000 IF SP = NS THEN GOTO 1090
1020 VTAB 22: PRINT "PRESS ANY KEY TO PLAY THE NEXT SONG.";
1030 GET A$: PRINT
1035 :
1036 :
1040 REM CLEAR ARRAYS TO AVOID REDIM ERR
1041 :
1042 :
1050 POKE 109,PEEK (107): POKE 110, PEEK (108)
1060 SP = SP + 1
1070 GOTO 480
1080 END
1085 :
1086 :
1090 REM END OF GAME
1100 :
1110 PRINT "THIS IS THE END OF THE GAME"
1120 PRINT
1130 PRINT "OUT OF ";NS * 15;" POSSIBLE POINTS, YOUR":
     PRINT "FINAL SCORE IS: ";SC
1140 PRINT
1150 IF SC < NS * 15 / 1.33 THEN GOTO 1190
1160 PRINT "YOU'RE FANTASTIC! WHAT AN EAR!!"
1170 PRINT "YOU REALLY KNOW YOUR MUSIC!"
1180 GOTO 1270
1190 IF SC < NS * 15 / 1.6 THEN GOTO 1230
1200 PRINT "YOUR SCORE IS NOT TOO BAD."
1210 PRINT "AFTER A LITTLE LISTENING. YOU ":
     PRINT "WILL DO BETTER!"
1220 GOTO 1270
1230 IF SC < ((NS * 15) / 2) - 1 THEN 1260
1240 PRINT "A SCORE BELOW 50% ISN'T VERY IMPRESSIVE!":
     PRINT "COME BACK AFTER YOU'VE LISTENED TO A FEW":
     PRINT "TAPES AND RECORDS."
1250 GOTO 1270
```

```
1260 PRINT "YOU'RE A DISGRACE TO MUSIC LOVERS EVERYWHERE!:
     PRINT "HAVE YOU EVER CONSIDERED REAL ESTATE??"
1270 END
1280 :
1290 :
1300 :
1310 REM TONE GENERATOR
1320 DATA 173,48,192,136,208,4,198,1,240,8,202,208,
          246,166,0,76,3,3,96
1330 FOR J = 771 TO 789: READ X: POKE J,X: NEXT J
1340 RETURN
1350 :
1360 FOR X = 1 TO N
1370 READ P,D
1380 POKE 0,P: POKE 1,D: CALL 771
1390 NEXT X
1400 RETURN
1410 REM
1420 REM
1430 REM REVEILLE
1440 DATA 255,60,192,60,154,30,192,30,255,60,
          154,60,192,60,154,30
1450 DATA 192,30,255,60,154,60,192,60,154,30,192,30
          255,60,255,60,192,255

1460 REM CHARGE!
1470 DATA 192,60,154,60,128,60,128,120,154,60,128,240

1480 REM TUT-DA-DA
1490 DATA 96,30,96,30,96,255

1500 REM DATA FOR FIRST TUNE
1505 :
1510 REM 1ST VALUE IS TOTAL NOTES IN THE SONG.
     THEN THE NAME OF SONG IS GIVEN.
1515 :
1520 DATA 10, "SOMEWHERE OVER THE RAINBOW"
1530 DATA 96,240,47,240,50,120,64,60,56,60,50,60,47,
     240,96,240,56,240,64,240
1535 :
1540 REM 2ND SONG
1545 :
1550 DATA 6,"DAY IS DONE"
1560 DATA 255,240,255,120,192,240,255,240,192,120,154,240
1565 :
1570 :
1580 :
1600 REM ADD MORE SONGS
```

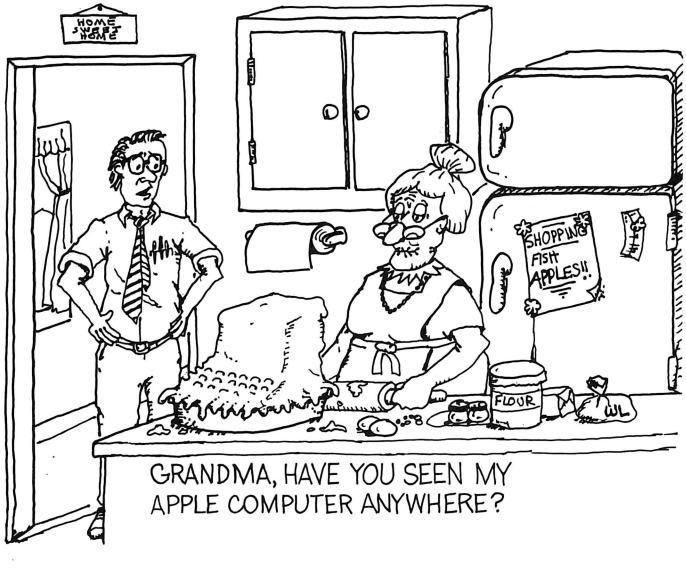
```
1620 REM IN LINE 1600 AND
1630 REM AND FOLLOWING.
1640 :
1650 :
1660 REM YOU MUST CHANGE
1670 REM THE VARIABLE NS IN
1680 REM LINE 40 TO EQUAL
1690 REM THE NUMBER OF SONGS
1700 REM YOU HAVE.
```

LISTING 14

The REM statements describe how this program works. Any programming techniques that you are not familiar with are explained in more detail elsewhere in this book. Check the table of contents or the index for their locations.

SPECIAL SOUND EFFECTS IN APPLESOFT BASIC AND INTEGER BASIC

The following program presents several exciting sound effects. Perhaps you may be able to use some of them in your own programs. Lines 38 through 49 list the sound demonstrations of this program.



```
 10 REM SOUND EFFECTS I
 15 FOR J = 771 TO 789
 17 READ D
 20 POKE J,D
 22 NEXT J
 25 DATA 173,48,192,136,208,4,198,1,240,8,202,208,
        246,166,0,76,3,3,96
 27 CALL -936
 30 X = 0
 32 POKE 32,6
 34 VTAB 2
 36 HTAB 11
 38 PRINT "WHICH DEMO ?"
 40 PRINT: PRINT: PRINT "1 UP UP AND AWAY"
 42 PRINT: PRINT "2 WHAT GOES UP MUST COME DOWN"
 43 PRINT: PRINT "3 FASTER UP UP"
 44 PRINT: PRINT "4 FASTER DOWN"
 45 PRINT: PRINT "5 DEATH RAY"
 46 PRINT: PRINT "6 DRIP"
 47 PRINT: PRINT "7 BIRD CHIRP"
 48 PRINT: PRINT "8 BIONIC CRICKET"
 49 PRINT: PRINT "9 EUROPEAN SIREN"
 50 PRINT
 55 INPUT C
 60 IF C > 10 THEN 27
 70 ON C GOTO C * 100

100 FOR T = 100 TO 1 STEP -1
110 POKE 0,T
120 POKE 1,20
140 CALL 2
150 NEXT T
160 GOTO 27

200 FOR T = 1 TO 100
210 POKE 0,T
220 POKE 1,50
240 CALL 771
250 NEXT T
260 GOTO 27

300 FOR T = 100 TO 1 STEP - 1
310 POKE 0,T
320 POKE 1,5
340 CALL 771
350 NEXT T
360 GOTO 27
```

```
400 FOR T = 1 TO 100
410 POKE 0,T
420 POKE 1,5
440 CALL 771
450 NEXT T
460 GOTO 27

500 FOR T = 100 TO 110
510 POKE 0,T
520 POKE 1,5
540 CALL 771
550 NEXT T
560 FOR T = 110 TO 100 STEP -1
570 POKE 1,1
580 CALL 771
585 NEXT T: X = X + 1
587 IF X >= 5 THEN 27
590 GOTO 500

600 FOR T = 15 TO 1 STEP - 1
610 POKE 0,T
620 POKE 1,3
640 CALL 771
650 NEXT T
660 GOTO 27

700 FOR J = 1 TO 3
705 FOR T = 1 TO 20
710 POKE 0,T
720 POKE 1,5
740 CALL 771
750 NEXT T : NEXT J
760 GOTO 27

800 FOR J = 1 TO 5
810 FOR Z = 1 TO 300: NEXT Z
820 FOR T = 1 TO 10
850 POKE 0,10
860 POKE 1,40
880 CALL 771
890 NEXT T
891 NEXT J
895 GOTO 27

900 FOR T = 1 TO 5
910 POKE 0,100 : POKE 1,150 : CALL 771
920 POKE 0,137 : POKE 1,200 : CALL 771
```

```
930 NEXT T
940 GOTO 27
```

Source: Modified from an INTEGER BASIC program from the Public Domain Library

These *Star Wars* sound effects use the same tone generator as SOUND EFFECTS I.

```
 10 REM SOUND EFFECTS II
 15 FOR J = 771 TO 789
 17 READ D
 20 POKE J,D
 22 NEXT J
 25 DATA 173,48,192,136,208,4,198,1,240,8,202,208,246,166,0
         76,3,3,96

 60 REM R2-D2
 70 HOME: VTAB 10: "PRINT "R2-D2"
 80 FOR G = 1 TO 15
 90 J = INT (RND(1) * 100) + 1
100 K = INT (RND(1) * 40) + 1
110 POKE 0,J
120 POKE 1,K
140 CALL 771
150 NEXT G
160 FOR J = 1 TO INT (RND(1) * 100) + 100
170 NEXT J

200 REM BLASTER
210 HOME: VTAB 10: PRINT "BLASTER"
215 FOR G = 1 TO 5
220 FOR J = 1 TO 28
240 POKE 0,J: POKE 1,5: CALL 771
250 NEXT J
260 NEXT G

300 REM EXPLOSION
310 HOME: VTAB 10: PRINT "EXPLOSION"
320 FOR J = 1 TO INT (RND(1) * 60)
330 X = INT (RND(1) * 50) + J
340 K = INT (RND(1) * 3) + 1
350 POKE 0,X: POKE 1,K: CALL 771
360 NEXT J

400 REM ALARM
410 HOME: VTAB 10: PRINT "ALARM"
420 FOR J = 1 TO 20: POKE 0,10: POKE 1,50: CALL 771
430 FOR J2 = 1 TO 25: NEXT J2,J
```

```
440 FOR J = 1 TO 100 : NEXT J

500 REM TIE FIGHTER
510 HOME: VTAB 10: PRINT "TIE FIGHTER"
515 FOR G = 1 TO 5
520 FOR J = 1 TO 10
530 X = J: K = 3
540 POKE 0,X: POKE 1,K: CALL 771
550 X = J * 2: K = 2
555 POKE 0,I: POKE 1,K: CALL 771
560 NEXT J
570 NEXT G
600 END
```

Source: Modified from an INTEGER BASIC program from the Public Domain Library

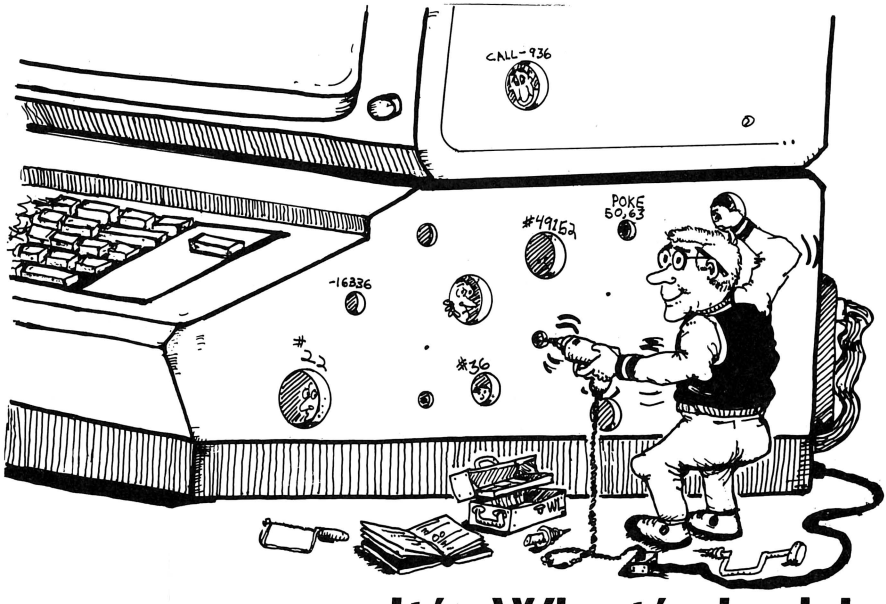
For other interesting sounds try this program written in *INTEGER BASIC*.

```
 10 REM SOUND EFFECTS III

 20 CALL -936: GOSUB 210
 30 POKE 1,160: POKE 3,165: POKE 4,0: POKE 5,41: POKE 6,31:
    POKE 7,24: POKE 8,105: POKE 10,32: POKE 11,168: POKE 12,252:
    POKE 13,173: POKE 14,48: POKE 15,192
 40 POKE 16,136: POKE 17,208: POKE 18,240: POKE 20,0: POKE 21,208:
    POKE 22,234: POKE 23,96
 50 POKE 2, RND (4) + 1: POKE 9, RND (8) + 7: POKE 19,198 +32 *
    RND (2): CALL 1

100 LET Z = Z + 1: IF Z > 4 THEN GOTO 200
110 GOTO 30

160 REM
170 REM SOUND EFFECTS
180 REM

200 POKE -16369,0: TEXT: CALL -936: VTAB 10: HTAB 10: PRINT "* * *
    THE END * * *": END
210 CALL -936: VTAB 5: HTAB 10: PRINT "SOUND EFFECTS":PRINT
220 PRINT:PRINT
230 HTAB 4: PRINT "THIS PROGRAM GENERATES SOUND": PRINT
240 HTAB 8: PRINT "EFFECTS (NO GRAPHICS)": PRINT
260 VTAB 21: PRINT " PRESS 'ESC' TO END"
270 VTAB 22: PRINT " PRESS THE SPACE BAR TO CONTINUE..."
280 IF PEEK (-16384) = 155 THEN 200
290 IF PEEK (-16384) < > 160 THEN 270
300 POKE -16368,0: TEXT: CALL -936: RETURN
```

Source: Public Domain Library

This program introduces a synthesizer effect.

```
 10 REM SPACE + INTRO
 20 :
 30 FOR K = 16128 TO 16251: READ J: POKE K,J: NEXT

 40 DATA 165,5,133,7,165,4,133,6,165,0,133,2,133,1,
         173,48,192,165,1,233,1,133,1,208,5,165,0,76,12,63,165,
         2,233,1,133,2,208,235,173,48,192,165,0,133
 50 DATA 2,165,7,233,1,133,7,208,220,165,5,133,7,165,
        6,233,1,133,6,208,208,96,165,0,133,4,165,1,133,5,165,2,
        133,6,165,3,133,7,173,48,192,165,5,233

 60 DATA 1,133,5,208,250,165,4,233,1,133,4,208,240,165,
         0,133,4,165,1,133,5,198,7,208,225,165,3,165,6,233,1,133,
         6,208,215,96

 70 M = 35:K = 250: GOSUB 240:K = 165: GOSUB 240:M = 120:K = 123:
    GOSUB 240
 80 :
 90 REM SEND TO SYNTHESIZER
100 :
110 M = 5:K = 250: GOSUB 200:M = 10:K = 165: GOSUB 200:M = 70:
    K = 123: GOSUB 200

120 M = 45:K = 250: GOSUB 240:K = 123: GOSUB 240:M = 90:K = 137:
    GOSUB 240:M = 12:K = 250: GOSUB 200:K = 123: GOSUB 200:
    M = 24:K = 137: GOSUB 200:

130 M = 45:K = 110: GOSUB 240:K = 137: GOSUB 240:K = 165: GOSUB
    240:K = 222: GOSUB 240:M = 20:K = 250: GOSUB 240:K = 165:
    GOSUB 240:M = 120:K = 123: GOSUB 240

140 K = 145: GOSUB 240:K = 128: GOSUB 240:K = 123: GOSUB 240:: END
150 :
160 :
170 REM SYNTHESIZER
180 :
190 :
200 Q = 32767 / (40 * K) * M: POKE 0,1 + ((K / 255 - INT (K / 255))
    * 255): POKE 1,K / 255 + 1: POKE 4,1 + ((Q / 255 - INT (Q /
    255)) * 255): POKE 5,Q / 255 + 1: CALL 16128: RETURN
210 :
220 REM PURE TONE
230 :
240 K = 26 * K / 19:Q = 32767 / (16 * K) * M: POKE 1,1 + ((K / 255
    - INT (K / 255)) * 255): POKE 0,K / 255 + 1: POKE 3,1 + ((Q /
    255 - INT (Q / 255)) * 255): POKE 2,Q / 255 + 1: CALL 16194:
    RETURN
```



2 It's What's Inside That Counts! Peeks, Pokes, and Calls

INTRODUCTION

What on earth are peeks, pokes, and calls? What do these strange words have to do with a computer?

> "I see them used in a lot of programs, but I don't know what they are or what they do!"

Well, they are not really very hard to understand or even to use. You can get more out of your Apple if you become familiar with its inner workings. Peeks, pokes, and calls refer to these inner mechanisms or memory locations which, when used properly, can save you a good deal of programming time.

To better explain them, let's use an analogy.

The Apartment Building

The computer is like a large apartment building. We call the apartment/memory location an address. Someone resides in each apartment. In the computer, the resident is a number.

In the case of an apartment tenant we may say,

"Mr. Johnson lives in apartment 222."

In the case of the computer we may say,

"5 resides in memory lcoation 222."

or

"The variable 5 resides at memory location or address 222."

Sometimes we don't know who lives in an apartment, but we can usually find out by looking at the apartment directory. Or maybe, as we pass by the apartment, its door may be open a little, and we can **PEEK** inside to see who's there.

If we want to find out which number resides at a particular memory location in the computer, we can **PEEK** inside.

For example, if we wanted to know what number resides in memory location 222, we could find out by typing

```
20 N = PEEK (222)
30 PRINT N
```

Line 20 tells the computer to **PEEK** at location 222 and make the number equal to the variable **N.** Line 30 tells the computer to print that number.

We could obtain the same information by typing

```
20 PRINT PEEK (222)
```

Sometimes in an apartment building, different people move in and out of the same apartment. One month you may find Mr. Jones in apartment 222.

The next time you visit, Mrs. Jackson may be living there instead.

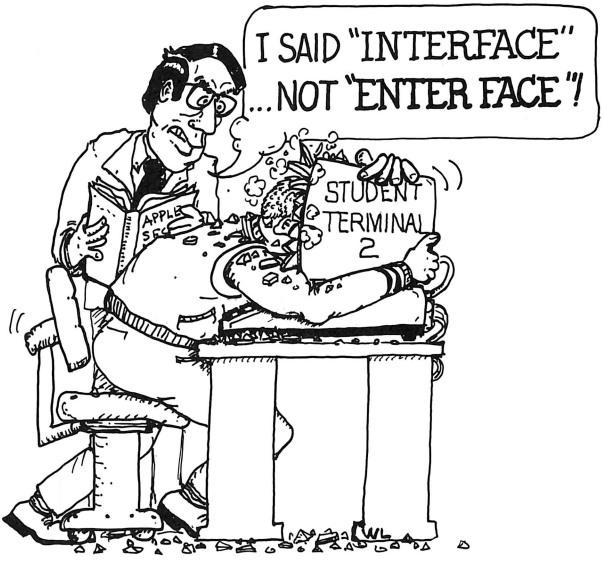
Numbers in the computer's memory locations can change as well. At one time, when we **PEEK** at memory location 222, it may contain 5. Another time, it may contain 16. Numbers can change depending on what happens during the program.

You are the manager of your computer's memory locations. You can **POKE** a new tenant (number) into a certain memory location. For example, to put 7 into memory location 34, we write

```
10 POKE 34,7
```

We **POKE** a number into a specific memory location because we want the computer to do something other than what if normally would do.

In the above example, we instructed the computer to change the top margin of the text window from 0 (the top of the screen) to 7 (the bottom of row 7). Unless



we change the **POKE** back to 34,0, the top margin will remain at the bottom of row 7.

Typing

```
60 PRINT PEEK (34)
```

would display 7.

Who's Calling My Number?

Often we want the computer to perform the same task more than once in a program. To avoid typing the same lines over and over, we can write a subroutine. Then, whenever the task needs to be performed, we simply send the program to the subroutine. The command for this is GOSUB. Every GOSUB has a RETURN at the end of the subroutine. RETURN tells the computer to go back to the next line after GOSUB.

The computer has built-in subroutines which we can access by using the **CALL** command.

For example, type

```
CALL -1184
```

to reveal a secret message on your screen.

One simple **CALL** command can perform routines that otherwise would take several program lines.

The following pages list **PEEKs, POKEs,** and **CALLs** and their functions. Some are cross-referenced to chapters in this book.

CHART OF PEEKs

	Paddle Control
PEEK-16284	PADDLE (3) BUTTON
PEEK-16285	PADDLE (2) BUTTON
PEEK-16286	PADDLE (1) BUTTON
PEEK-16287	PADDLE (0) BUTTON

Example
```
10 X = PEEK (-16287)
20 IF X > 127 THEN . . . (Button pressed)
30 IF X < 127 THEN . . . (Button not pressed)
```

PEEK-16336 CLICK SPEAKER (see Chapter 1)

Example
```
10 S = PEEK (-16336)
```

PEEK-16352 Toggles cassette output once. Produces a "click" on a cassette recording

MARGINS *(see Chapter 3)*

PEEK 32	Left margin of text window (0–39)
PEEK 33	Width margin of text window (1–40)
PEEK 34	Top margin of text window (0–23)
PEEK 35	Bottom margin of text window (1–24)

CURSOR LOCATIONS *(see Chapter 3)*

PEEK 36	Horizontal cursor position (0–39) CH = PEEK (36)
PEEK 37	Vertical cursor position (0–23) CV = PEEK (37)

PRINT PEEK (105) + PEEK (106) * 256
Shows the current value of LOMEM

PRINT PEEK (115) + PEEK (116) * 256
Shows the current value of HIMEM

ERROR RELATED *(see Chapter 6)*

PRINT PEEK (123) + 256 * PEEK (124)
Prints the current data line being read

PEEK 216	ONERR flag
PEEK 218, 219	Line where ONERR error occurred
PEEK 222	ONERR error code
PEEK 978	Print memory size if used as:

```
PRINT (PEEK (978) + 35)/4;"K"
```

Chart of Peeks (cont.)

PEEK 2040 Determines which slot is active (0 is active if no peripheral device has been activated).

```
SLOT = PEEK (2040) - 192
PRINT SLOT
```

PEEK 49152 Reads keyboard (see Chapter 5)

PEEK 49200 Clicks speaker

```
S = PEEK (49200)
```

CHART OF POKEs

Annunciator Outputs

POKE-16295	Output 0 On (TTL low, 0.3 volt)
POKE-16293,0	Output 1 On
POKE-16291,0	Output 2 On
POKE-16289,0	Output 3 On
POKE-16296,1	Output 0 On (TTL high, 3.5 volts)
POKE-16294,1	Output 1 On
POKE-16292,1	Output 2 On
POKE-16290,1	Output 3 On

Annunciator outputs are built into the game paddle connector. You could use LED indicators to indicate whose turn it is, etc. (See Apple Reference Manual for more detail.)

Graphic Pokes

POKE-16297,0	hi-res
POKE-16298,0	Lo-res
POKE-16299,0	Shows page 2
POKE-16300,0	Shows page 1
POKE-16301,0	Split-graphics and 4 text lines
POKE-16302,0	Shows full-screen graphics
POKE-16303,0	Shows current text screen
POKE-16304,0	Shows current graphics screen

POKE-16368,0 CLEAR KEYBOARD STROBE (see Chapter 5)

Prevent Language Card Erasure

POKE-16427,234 (normal: 141)
POKE-16428,234 (normal: 0)
POKE-16429,234 (normal: 224)

These three POKEs prevent erasure of the language card when the new disk is booted.

Technique

1. Boot System Master.
2. Make POKEs.
3. Initialize a new disk.
4. New disk will then boot without erasing the language card.

POKE 33,33 SPECIAL POKE FOR EDITING AND MORE (See Chapters 3 and 7)

Cursor Location *(see Chapter 3)*

POKE 36,CH	Returns current horizontal position of cursor
POKE 37,CV	Returns current vertical position of cursor
POKE 50,63	INVERSE
POKE 50,127	FLASH
POKE 50,128	Makes listings and catalog invisible (normal value: 255)
POKE 50, random	Scrambles text output
POKE 50,255	NORMAL
POKE 82,128	Makes cassette program auto-RUN when loaded
POKE 109, PEEK (107): POKE 110, PEEK (108)	Deletes all arrays. Lets you redimension any or all arrays without error.
POKE 214,128 or POKE 214,255	Make disk program RUN when any command is issued (normal value: 0)
POKE 216,0	Cancel ONERR (see Chapter 6)
POKE 241,1	Sets SPEED to 255. The second number is equal to 256-SPEED.
POKE 1010,102: POKE 1011,213: POKE 1012,112	Makes RESET RUN. Pressing RESET unhooks DOS. Use CALL 1002 at the beginning of the program to rehook (normal values: 191,157,56).
POKE 1011,0: POKE 1012,0 or POKE 1011, 224	Causes computer to reboot when RESET is pressed.

Ampersand Vector or the "And Hook"
(See Chapter 1 for more detail)

POKE 1013,76: POKE 1014,26: POKE 1015,252	Makes "&" move cursor up one line
POKE 1013,76: POKE 1014,58: POKE 1015, 255	Makes "&" ring bell
POKE 1013,76: POKE 1014,66: POKE 1015, 252	Makes "&" clear to end of page
POKE 1013,76: POKE 1014,88: POKE 1015, 252	Makes "&" = HOME
POKE 1013,76: POKE 1014,156: POKE 1015, 252	Makes "&" clear to end of line
POKE 1014,110: POKE 1015,165	Makes "&" CATALOG disk
POKE1014,165; POKE 1015,214	Makes "&" LIST
POKE 2049,1	Prevents program from LISTing once it's been RUN.
POKE 40286,252: POKE 40287,164	Makes RESET RUN
POKE 44452,22: POKE 44605,21	Controls the number of files displayed in CATALOG. The first POKE is always 1 more than the second POKE.
POKE 44505,234: POKE 44506,234	Reveals deleted files in your catalog (normal values: 48,74)

Chart of Pokes (cont.)

POKE 44509,76: POKE 44510, 19: POKE 44511, 174	Shows only file names in CATALOG. Omits file type and sector length. (Normal values 189,200, 180)
POKE 44545, 169: POKE 44546, X POKE 44547,234	where x = 0 to 255. Shows sector length of all files in CATALOG to be "x" length. (Normal values: 189,231,180)
POKE 44611,1	Changes catalog sector number from 3 to 2
POKE 45620,234: POKE 45621,234	Keeps repeating first file in CATALOG (normal value: 105,35)
POKE 49168,0	Clears keyboard strobe (see Chapter 5)

Graphic Pokes

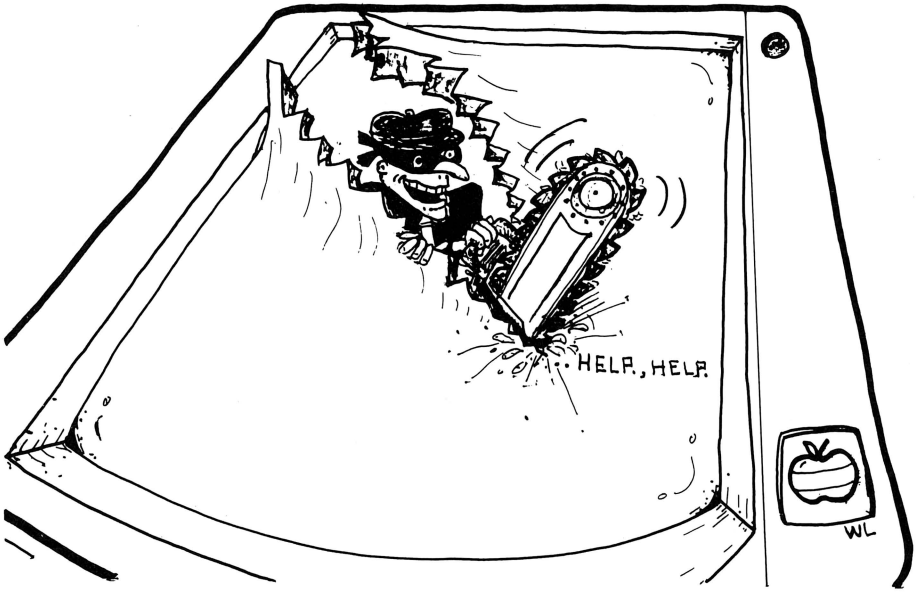
POKE 49232,0	Shows current graphic screen
POKE 49233,0	Shows current text screen
POKE 49234,0	Shows full-screen graphics
POKE 49235,0	Shows split-graphics and 4 text lines
POKE 49236,0	Shows page 1
POKE 49237,0	Shows page 2
POKE 49238,0	Lo-res
POKE 49239,0	Hi-res
POKE 49385,0	Spins disk without reading, writing, or error
POKE 49384,0	Stops drive

CHART OF CALLs and WAITs

CALL -144	Scans input buffer
CALL -151	Enters monitor, no bell
CALL -155	Enters monitor, with bell
CALL -167	Returns to monitor in text mode, same effect as pushing the RESET button.
CALL -198	Rings bell (see Chapter 1)
CALL -211	Prints "ERR" and rings bell
CALL -380	NORMAL
CALL -384	INVERSE
CALL -550	Prints hex value of A-register
CALL -657	Inputs (commas and colons ok)
CALL -676	Rings bell and waits for carriage return
CALL -756	Waits for keypress
CALL -856	Creates a short wait loop
CALL -868	Clears current text line from the cursor to the right (see Chapter 3)
CALL -875	Clears entire text line
CALL -912	Scrolls up one line
CALL -922	Moves cursor down. Issues a line feed.
CALL -936	Clears text (HOME) (see Chapter 3)
CALL -958	Clears text cursor to bottom (see Chapter 3)
CALL -998	Moves cursor up
CALL -1008	Moves cursor left
CALL -1036	Moves cursor right
CALL -1052	Rings bell

Chart of Pokes (cont.)

CALL -1184	PRINTs "Apple II" (newer Apples)
CALL -1205	Sets normal text window
CALL -1216	GR
CALL -1233	TEXT
CALL -1321	Displays all registers
CALL -1728	Prints hex of X and Y registers
CALL -1953	Adds 3 to COLOR
CALL -1994	Clears upper lo-res to black
CALL -1998	Clears full lo-res screen to black
CALL -2613	Hi-res cursor coordinates to 224–226
CALL -3082	Clears hi-res to recent HCOLOR
CALL -3086	Clears hi-res screen to black
CALL -3106	HGR
CALL -3116	HGR2
CALL 768	Often used in clearing up ONERR GOTO problems (see Chapter 6)
CALL 1002	Rehook DOS
CALL 11246	Clears HGR2 screen to black
CALL 11250	Clears HGR2 screen to the HCOLOR last HPLOTed
CALL 42340	Locks up disk to prevent writing on it (**CAUTION!:** before using, back up the disk you are using)
CALL 42350	CATALOG
CALL 62450	Clears current hi-res screen to black
CALL 62454	Clears current hi-res screen to HCOLOR most recently HPLOTed. Must be preceded by a HPLOT.
WAIT -16384,128	Computer will wait until key is pressed.



3 The Text Window Massacre

DAZZLING SCREEN DISPLAYS

The portion of the screen where text or high- and low-resolution graphics is printed is called the *text screen window*. You can manipulate the text window in many ways to create exciting displays. Chapter 4 gives several examples of creative things you can do to enhance an ordinary program.

When we want to use only a particular section of the screen, we must inform the computer. We can do this by POKEing screen positions into the computer's memory. These can be top, bottom, or left margins, or width.

For example, suppose you want to protect information in the top half of the screen from scrolling out of sight after the screen is filled. To do this you could enter

```
210 POKE 34,12
```

Any PRINT statements that follow this command will position the text below line 12. When the text has filled the bottom half of the screen, the top line (now line

12) will scroll off the screen. Any text above line 12 remains intact and unaffected. To restore the top margin back to the top of the screen (its normal position), type

```
290 POKE 34,0
```

The first number in the POKE tells the computer what margin you want to change. The second number is the margin's new position.

Figure 3-1 is a diagram of the screen with the margins marked.

Let's discuss how to control and change each margin:

Left Margin

Change the left margin with

```
10 POKE 32,L
```

where L is equal to a number from 0 to 39. The left margin of a normal text window is POKE 32,0.

Width

Change the width (number of characters per line) with

```
20 POKE 33, W
```

where W is equal to a number from 1 to 40.

IMPORTANT: Never use 0 or you will bomb Applesoft!

The normal width is POKE 33,40.

IMPORTANT: The sum of the window width and the leftmost window position should never exceed 40; that is,

$$L + W <= 40$$

If the sum is greater than 40, then characters may be placed in memory locations that will endanger your program or data. The width of the window is not automatically changed when you change the left margin (POKE 32, L). It is best to reduce the window width first, and then change the left margin. For example,

```
10 POKE 33, 36 : POKE 32, 4
```

Note that **36** + **4** = 40. The POKEs will provide a new margin without error or problems.

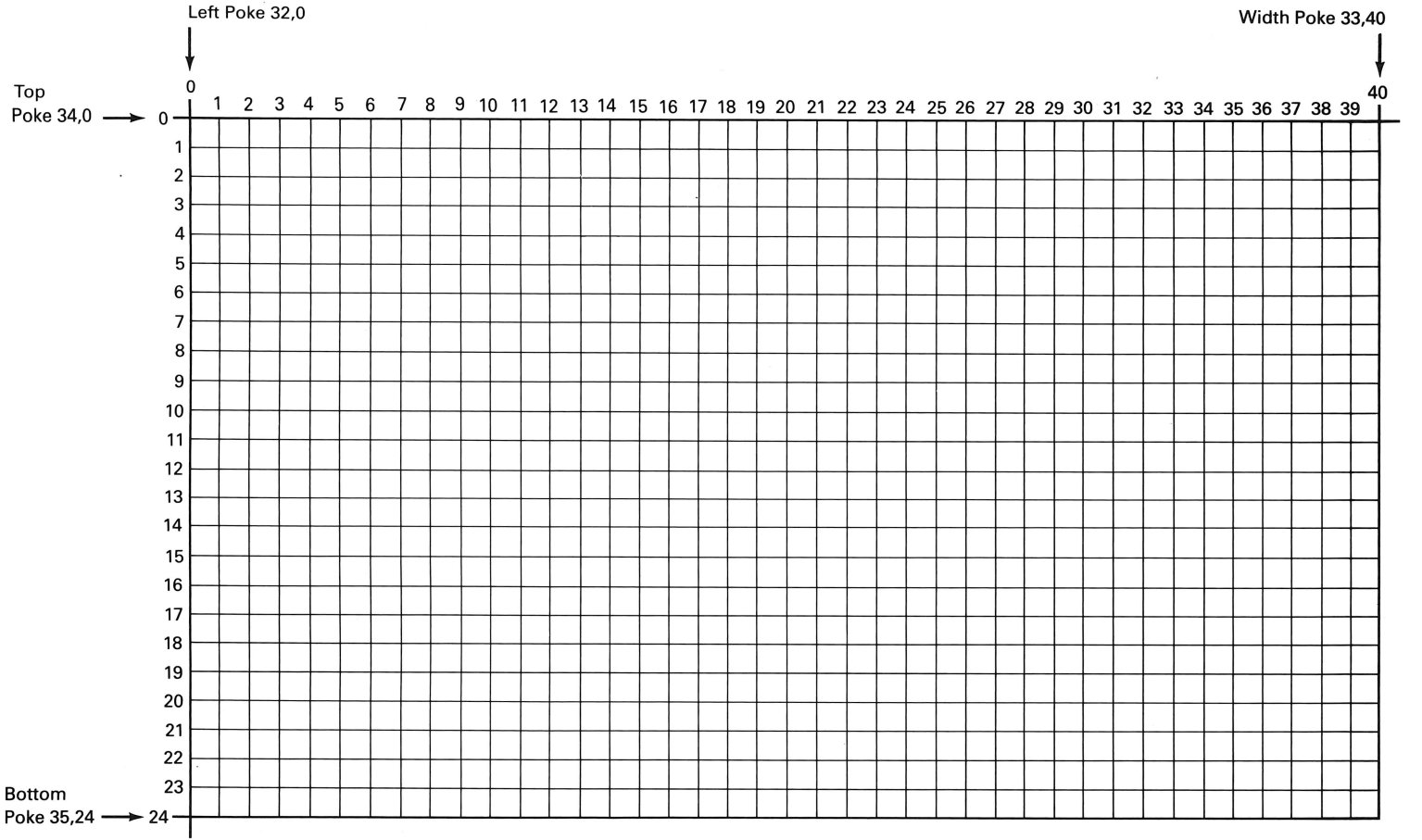


Figure 3-1 Margin Grid.

Top Margin

Change the top margin with

```
30 POKE 34, T
```

where T is equal to a number from 0 to 23. The normal top margin position is 0; i.e., POKE 34,0. Never set the top margin lower than the bottom margin.

Bottom Margin

```
40 POKE 35, B
```

where B is equal to a number from 0 to 24. The normal bottom margin position is POKE 35, 24. Never set the bottom margin higher than the top margin.

TWENTY-FOUR LINES OF INFORMATION???!

The APPLE can display 24 lines of 40 characters each. However, if you have ever tried to write a program that displays all 24 lines at once, you know that it is difficult. Try RUNning this program:

```
10 HOME
20 FOR L = 1 to 24
30 PRINT L
40 NEXT L
50 END
```

What happened? When the program finished executing, the top two lines disappeared. The cursor had to find a place to rest. The END in line 50 causes a RETURN and line feed.

If we add line 45,

```
45 VTAB 23
```

we recover one of the lines by instructing the cursor to jump up one line. However, we still haven't displayed all 24 lines at once.

By rewriting the program to halt output to the screen, we can display all 24 lines:

```
10 HOME
20 FOR L = 1 to 23
30 PRINT L
40 NEXT L
```

```
50 PRINT L;
60 CALL -756
70 END
```

Lines 20–40 create a loop to print numbers from 1 to 23. The twenty-fourth number is printed in line 50. The semicolon prevents the line feed. Line 60 temporarily halts output to the screen until a key is pressed. When the key is pressed, the prompt reappears, scrolling away line 1.

Although this demonstrates how to print all 24 lines, in practice you will rarely, if ever, need to do this. But if you do, now you know how!

CONTROLLING THE CURSOR

The standard Apple screen has 40 horizontal positions (columns) and 24 vertical positions (rows). We may position the cursor anywhere on the screen by using **HTAB** and **VTAB.**

HTAB positions the cursor in a particular column. For example,

```
10 HTAB 15: PRINT "NAME: "
```

moves the cursor to column 15 of the current line and prints NAME:. Since a line consists of only 40 columns, if we type

```
10 HTAB 43
```

the cursor will move down one row and right three columns. That is, the cursor moves right to column 40, runs out of room, wraps around (moves down to the next row), and continues another three columns.

VTAB positions the cursor in a particular row (up or down). For example,

```
10 VTAB 9
```

moves the cursor down to row 9. The rows are numbered from 1 to 24, as shown in Figure 3-2.

HTAB and VTAB together will position the cursor in any column or row desired. For example,

```
10 VTAB 9: HTAB 15: PRINT "HELLO"
```

prints HELLO in row 9, beginning at column 15.

VTAB ignores any text window you may have set using POKES 32 through 35 (see the section on text windows earlier in this chapter). VTAB uses absolute moves, relative only to the top and bottom of the screen.

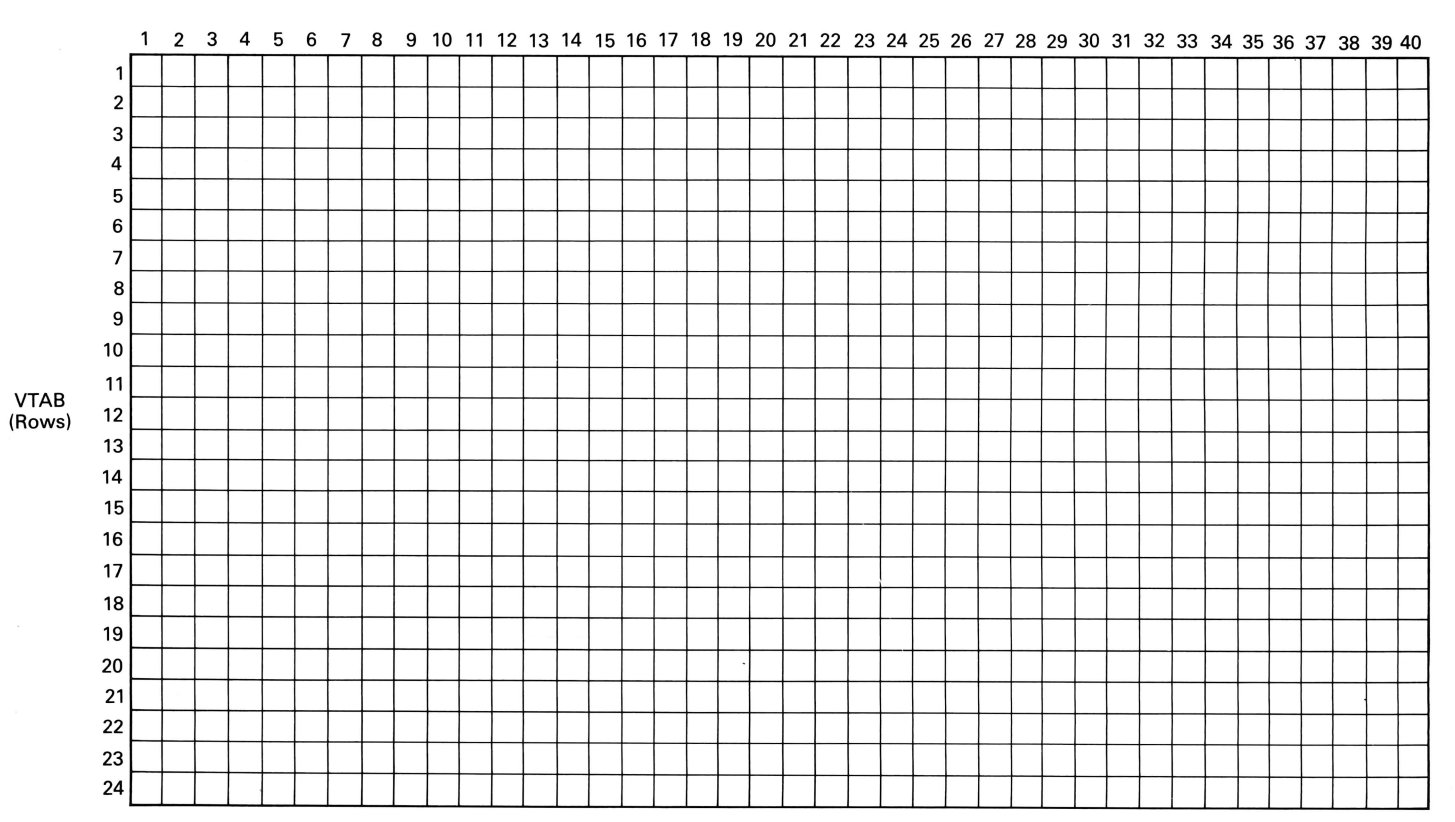


Figure 3-2 HTAB/VTAB Grid.

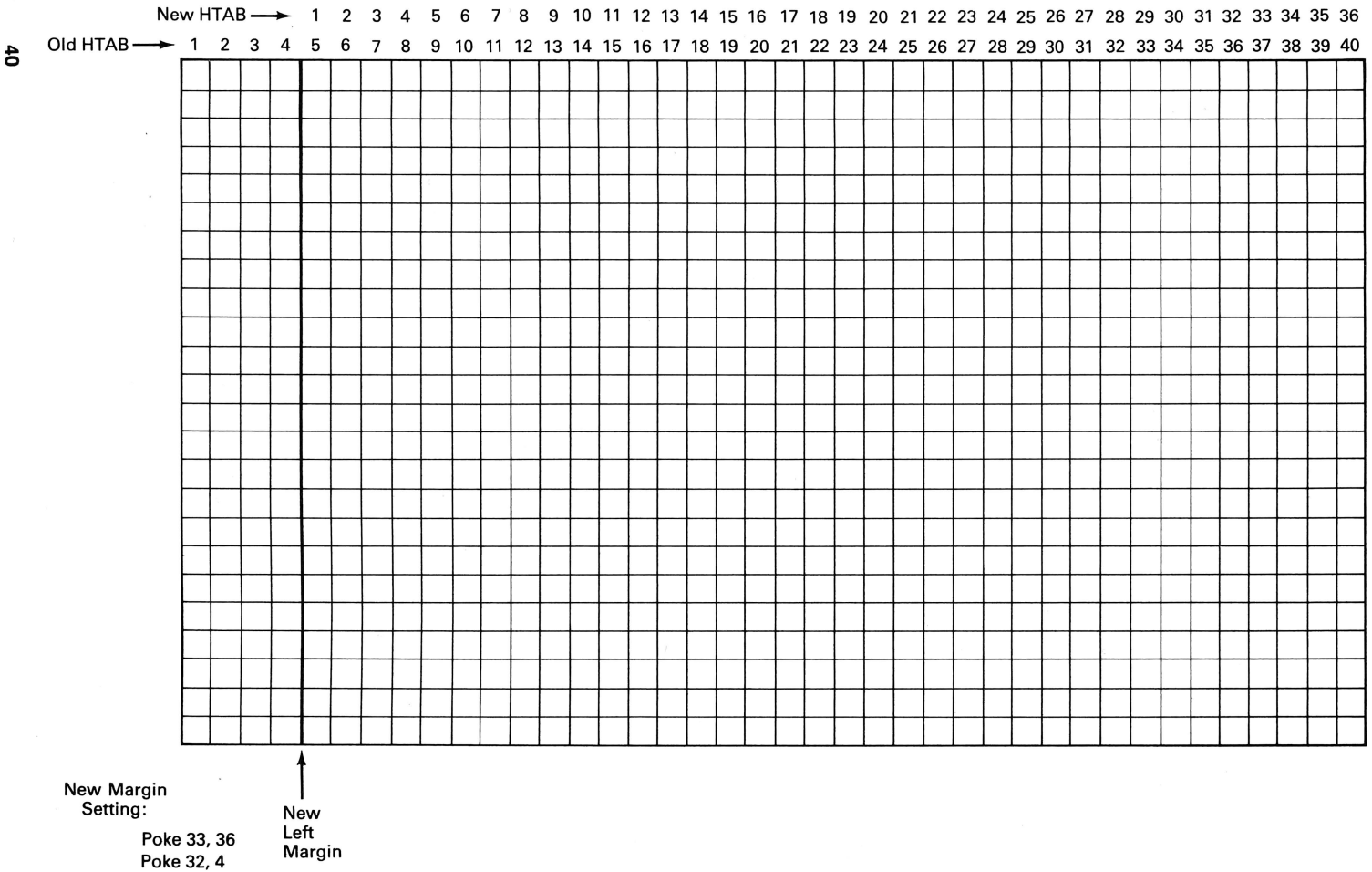


Figure 3-3 New HTAB/Old HTAB.

HTAB positions are relative to the left margin of the text window. For example, if you POKE 32,4, the left margin will move to column 4. See Figure 3-3.

If we assume that the left margin is at column 4, typing

```
20 PRINT "A"
```

puts A in what used to be column 5. If you type

```
20 HTAB 2: PRINT "A"
```

the A appears in what used to be column 6.

We see that using HTAB 1 places the cursor in the leftmost position of the current line of the text window.

SPC

Another valuable command is SPC (aexpr). The variable "aexpr" can be equal to any number from 1 to 255. Whereas ". . . HTAB moves the cursor to an absolute screen position relative to the left margin of the text window, SPC (aexpr) moves the cursor a given number of spaces away from the previously printed item."[1]

Suppose you want to print two words with 15 spaces between them. You could, for example, type

```
10 PRINT "JOHN               FRED"
```

One problem with this method is making sure you type exactly 15 spaces. A better way, using SPC, is

```
10 PRINT "JOHN";SPC(15);"FRED"
```

REMEMBERING WHERE THE CURSOR IS (PEEK AND POKE 36 and 37)

Let's explore other ways to control cursor movement. The computer can store the cursor's current location (HTAB or VTAB) in its memory. This is often very important in programming.

We can assign any variable we choose to the cursor's current horizontal or vertical position. Later we can easily send the cursor back to that location. For example, a program may ask for entry of a name and address.

[1]Copyright 1978, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

```
10 INPUT "ENTER NAME: ";N$
20 INPUT "ENTER ADDRESS: ";AD$
30 INPUT "ENTER CITY: ";CT$
40 INPUT "ENTER STATE: ";ST$
50 INPUT "ENTER ZIP: ";ZP$
```

If we RUN the above program, only ENTER NAME: appears on the screen initially. The user must respond before the next prompt ENTER ADDRESS: appears, and so on.

It would be more efficient to put all the prompts on the screen at the same time, as in Figure 3-4.

The user could answer a prompt and simply press RETURN to move to the next. We can easily create screen displays like these with POKE and PEEK 36 and 37. We will give examples of how to use these in programs, but first, let's discuss how they work.

CH = PEEK (36) saves and assigns the current horizontal position of the cursor to the variable CH.

CV = PEEK (37) saves and assigns the current vertical position of the cursor to the variable CV.

PEEK (36) counts columns from 0 to 39, and **PEEK (37)** counts rows from 0 to 23. This is unlike HTAB, which counts columns from 1 to 40, and VTAB, which counts rows from 1 to 24. See Figure 3-5.

As with HTAB, PEEK (36) positions are relative to the left margin of the

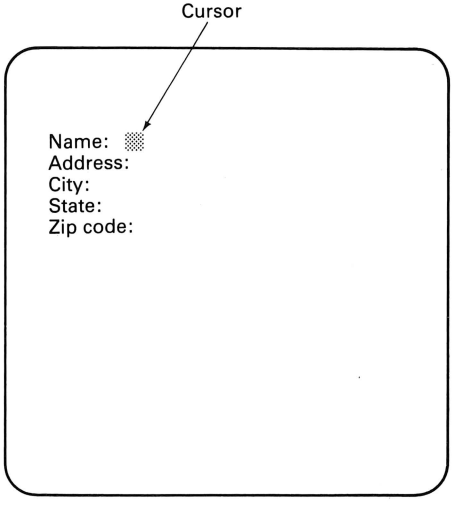


Figure 3-4 Screen Display With Cursor.

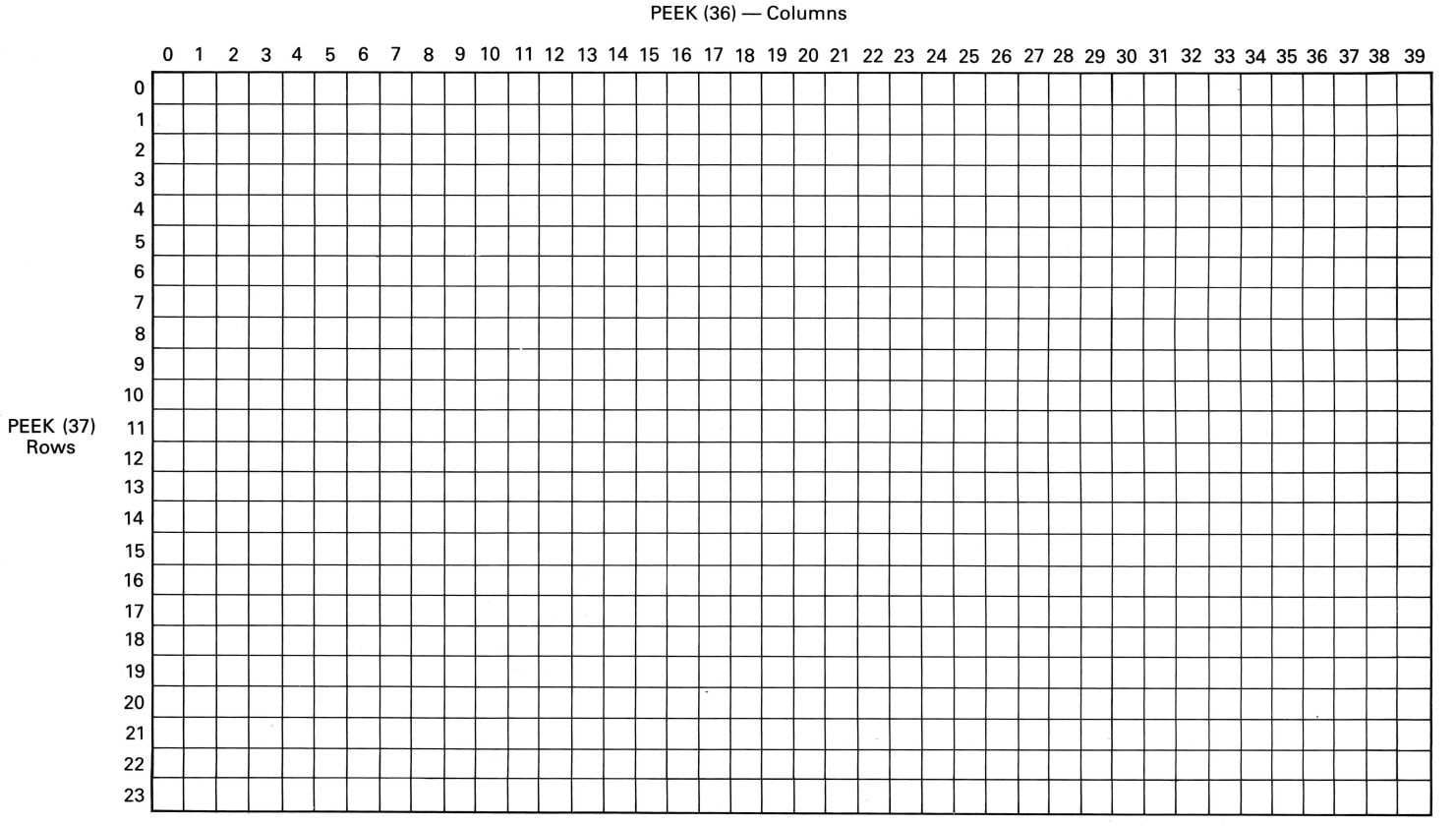


Figure 3-5 PEEK 36 & 37 Grid.

text window. However, because of the way PEEK (36) counts columns, the leftmost position of the current line in the normal text window is 0, not 1.

In the diagram in Figure 3-6, the cursor appears in position 3 from the left-hand edge of the screen. To see the cursor's position in Figure 3-6, type

```
20 CH = PEEK (36)
30 PRINT CH
```

The PEEK (36) position is 2, the value assigned to the variable CH.

Suppose you later decide to use CH in an HTAB statement to position the cursor at the same location. If, for example, you type

```
40 HTAB CH: REM CH = 2
```

you will be disappointed. The cursor will be printing in position 2, not 3 as you expected. Remember, PEEK (36) position is different from printing position.

Fortunately, there is a way to overcome the confusion between counting columns by HTAB and PEEK (36). Just type

```
20 CH = PEEK (36) + 1
```

Later, if we use

```
40 HTAB CH: PRINT "ANYTHING"
```

the characters will be positioned correctly.

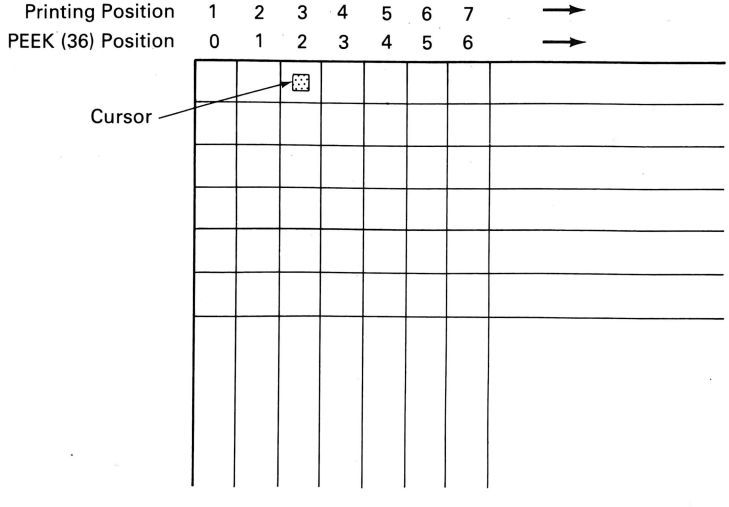


Figure 3-6 PEEK 36 Printing Position.

To solve that problem, we had to add 1 to the PEEK (36) value to ensure correct repositioning. The Apple will automatically add 1 to PEEK if we use, for example, POKE 36, CH.

This command will move the cursor to position CH + 1 ". . . printing-positions from the left margin to the text window."[2] Therefore, we could write a program as follows:

```
 5 HOME
10 VTAB 5
20 PRINT "NAME: ";:CH = PEEK (36)
30 PRINT
40 N$(1) = "WAYNE": N$(2) = "SALLY": N$(3) = "GEORGE"
50 FOR X = 1 TO 3
60 VTAB 5
70 POKE 36, CH: PRINT N$(X)
80 FOR J = 1 to 1000: NEXT J: REM TIME DELAY
90 NEXT X
100 END
```

Type the program and see what it does. We could get the same results by changing lines 20 and 70 to

```
20 PRINT "NAME: ": CH = 7
70 HTAB CH: PRINT N$(X)
```

But the disadvantage is that when we use HTAB this way, we have to do the counting ourselves to determine the column number. It is much easier to let the computer do it with PEEK (36) and POKE (36).

Setting Limits on Inputs

Sometimes you must limit the number of characters in an INPUT response. Otherwise, you may get unexpected results or an error message. One way to instruct the user is to include the limits in the INPUT statement, as follows:

```
10 INPUT "ENTER NAME (10 CHAR. MAXIMUM): ";N$
20 IF LEN (N$) > 10 THEN GOTO 10
```

The following program shows another clever way to use PEEK (36).

```
10 HOME
20 VTAB 10: PRINT "ENTER STOCK NUMBER: -----";: HTAB
   (PEEK (36) - 4)
```

[1]Copyright 1978, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

```
30 FOR K = 1 to 5: GET N$: IF N$ = CHR$(13) THEN PRINT:
   PRINT "ENTRY ERROR!":FOR J = 1 to 2000: NEXT J: SN$ = "":
   HOME: GOTO 20
40 PRINT N$;: SN$ = SN$ + N$: NEXT K: PRINT
```

Line 20 uses dashes to show how many characters can be entered. After the computer prints the fifth dash, the cursor stays at that position because of the semicolon. PEEK (36) − 4 then moves it left four columns so that the user can enter the part number "on top" of the dashes.

Line 30 uses a FOR NEXT loop to enforce the five-character limit. Line 30 also contains an error trap. When all the characters have been entered, the program continues. Line 40 prints the result. **PRINT N$;** moves the cursor right, to the next position, rather than down to the next row.

You could protect this program by requiring the user to enter a code first. The following example shows how. For added security, line 40 prints an "X" in place of the code you type (in case someone is looking over your shoulder).

```
10 HOME
20 VTAB 10: PRINT "ENTER A SECURITY CODE: -----;:
   HTAB (PEEK (36) − 3)
30 FOR K = 1 TO 4: GET A$: IF A$ = CHR$(13) THEN
   PRINT: CALL −198: PRINT "ENTRY ERROR!": FOR J = 1
   TO 2000: NEXT J: SC$ = "": HOME: GOTO 20
40 PRINT "X";: SC$ = SC$ + A$: NEXT K: PRINT
50 IF SC$ <> "CODE" THEN PRINT: CALL −198: CALL −198:
```

```
        PRINT "ACCESS DENIED!": FOR J = 1 to 2000: NEXT J:
        SC$ = "": HOME: GOTO 20
60  REM REST OF PROGRAM
```

If you have a top-secret program or if you're just paranoid, you can protect your program further by disabling the RESET and CONTROL-C commands (see Chapters 6 and 7).

Note: "CODE" in line 50 can be equal to anything you want, i.e., numbers, letters, a word, etc.

PEEK (37) and POKE (37) (aexpr)

We have already learned that CV = PEEK (37) saves and assigns the current vertical position of the cursor to the variable CV.

As with PEEK (36), which counts columns differently from HTAB, PEEK (37) counts rows differently from VTAB. Whereas VTAB counts rows from 1 to 24, PEEK (37) counts them from 0 to 23. See Figure 3-7.

Thus in CV = PEEK (37), CV = 0 is the top row of the screen and CV = 23 is the bottom.

As with VTAB, PEEK (37) also ignores any text window you may have set. Its absolute vertical position is relative only to the top and bottom of the screen.

In the example in Figure 3-8, the cursor is at VTAB 3, but its PEEK (37) position is 2.

We can illustrate the use of PEEK (37) with a contrast. The following program shows handling errors:

```
10  INPUT "NAME: ";N$
20  IF N$ = "" THEN 10
```

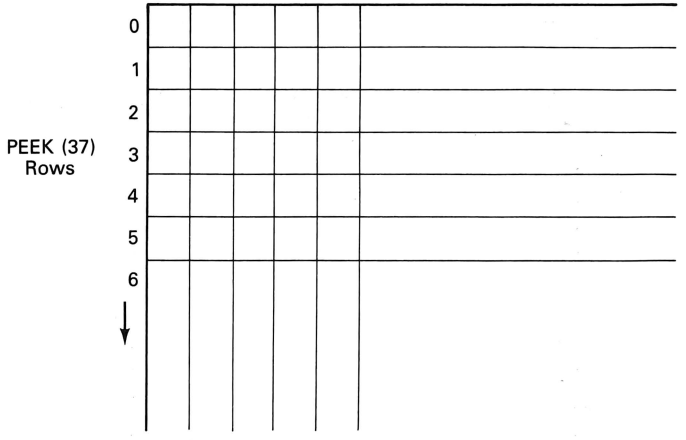


Figure 3-7 PEEK 37 Grid.

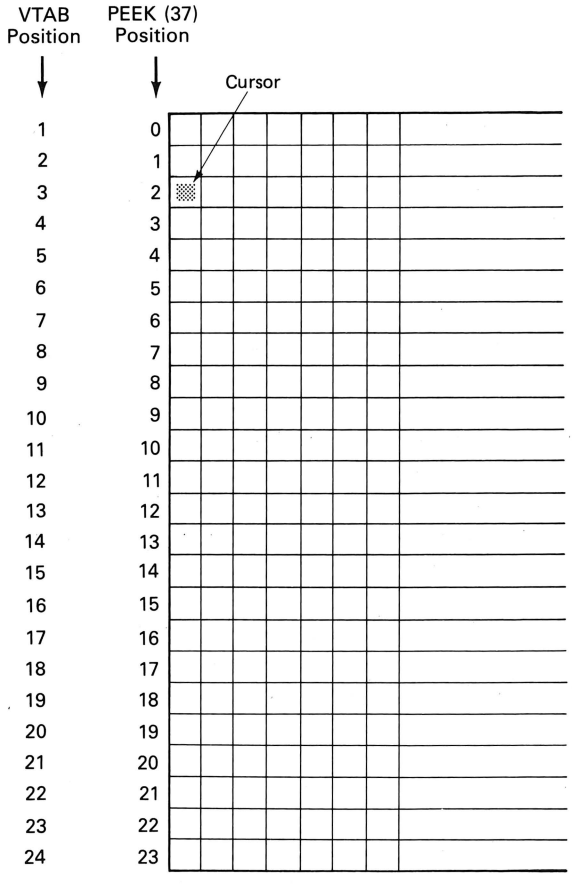


Figure 3-8 VTAB/PEEK (37)Grid.

If the user fails to enter anything, line 20 will catch the error and send the program back to line 10. Although this does what we intend, NAME prints twice as follows:

```
NAME:
NAME:
```

You can avoid this by changing lines 10 and 20 to

```
10 INPUT "NAME: ";N$
20 IF N$ = "" THEN VTAB PEEK (37): HTAB 1: GOTO 10
```

PEEK (37) saves the current vertical position. VTAB keeps the cursor on the same line. HTAB 1 instructs the computer to print the statement again in the first column

of the text window. The cursor appears to stay still, giving a more professional-looking screen effect.

If you want to ensure that only numbers can be entered, type

```
10 INPUT "PRICE OF ITEM: ";PC$
20 IF VAL (PC$) < = 0 THEN VTAB PEEK (37): CALL -958: GOTO 10
```

Again, PEEK (37) saves the current vertical tab position. CALL −958 clears the screen from the cursor down. The incorrect response is erased, and the cursor waits at the original location for another entry.

Here's a way to use PEEK (37) and also provide the user with further instructions.

```
100 PRINT "ENTER YOUR NAME: ": CV = PEEK (37): VTAB 22: HTAB 5:
    FLASH: PRINT "NO MORE THAN 10 CHARACTERS":
    NORMAL: VTAB CV: HTAB 18: INPUT ""; N$: CALL -958:
    IF N$ = " " THEN VTAB CV: HTAB 1: GOTO 100
```

Note the additional message or instruction at the bottom of the screen. After the user has entered something, CALL −958 erases the message and then traps the error.

We should mention three other CALL commands that affect screen displays.

HOME IS WHERE I BELONG (CALL −936)

CALL −936 is just another name for an old friend, the HOME command. You may use either one to clear the entire text window and send the cursor immediately to the top leftmost printing position.

WATCH OUT BELOW! (CALL −958)

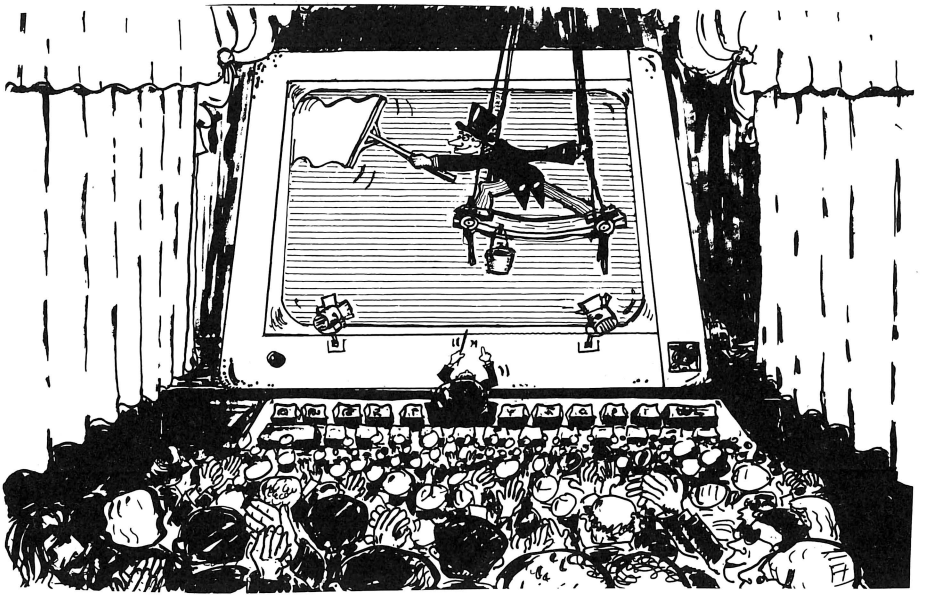
CALL −958 erases everything from the cursor position to the bottom margin. Everything above or left of the cursor remains intact and unaffected. Examples of the use of CALL −958 can be found in Chapter 4.

ALL RIGHT! OUT OF SIGHT! (CALL − 868)

CALL −868 clears everything from the cursor to the right margin of the current line. CALL −868 is a quick way to clear a line of unwanted text, as shown below:

```
10 HOME
20 PRINT "A QUICK DISAPPEARING ACT"
30 PRINT "CAN BE MADE . . . IF YOU KNOW THE SECRET!"
40 FOR X = 1 to 2000:NEXT X: REM TIME DELAY
50 VTAB 1:HTAB 1:CALL - 868: VTAB 3
60 END
```

See Chapter 4 for several examples of the use of CALL −868.



4 Clean Screens for the Cinematic Fanatic

INTRODUCTION

LIGHTS! CAMERA! ACTION!

SCENE 1, TAKE 2!

As the scene progresses, tension builds in the audience.

READY CAMERA TWO?

FADE TO TWO!

A new scene unfolds on the TV screen.

Whether staged on television or at the movies, an exciting cinematic screen clear adds variety and dramatic impact. It can blend scenes together or can abruptly change the tempo and mood.

Just as in those visual arts, watching programs evolve on the computer screen

can be exciting and fun to watch. There are many ways to move from "scene" to "scene."

The most common way to clear the screen is

```
10 HOME
```

HOME completely clears the screen and sends the cursor to the upper left corner of the screen or text window.

It is a good idea to start each program with this command. That way you start with a **clean screen.**

Most programmers use HOME several times in a program. Although it gets the job done, it's not too exciting to watch. There are many more creative ways to clear the screen.

In this chapter we will use several examples of screen clears. When choosing a screen clear, always keep in mind what is on the screen. For example, you wouldn't want a screen clear that erases from the center out if the display consists of only two or three lines at the top.

The routine in lines 1000–1070 in the following program fills the screen with asterisks. We will call this our Fill routine and use it throughout the chapter to demonstrate different screen clears.

Note that we use high line numbers to distinguish the Fill routine from the rest of the program. Note also the advantages of using END in lines 999 and 1070.

1. You can type RUN 1000 to run the Fill routine by itself. The END in line 1070 stops the program when the screen is full.
2. Typing RUN by itself runs the entire program. The END in line 999 keeps the program from returning to line 1000 for a "refill."

To type a different screen clear program, simply delete everything up to line 1000. In our example, you would type DEL 10,70. This would leave the Fill routine intact and ready to use with the next screen clear program.

```
 10 REM SCREEN CLEAR PROGRAM
 20 :
 30 REM     REST OF PROGRAM
 40 REM        GOES HERE
 50 :
 60 :
 70 :
999 END
1000 REM FILL ROUTINE
1010 HOME
1020 F$ = "****************************************"
      : REM 40 ASTERISKS
1030 FOR J = 1 TO 23
1040 VTAB J: HTAB 1
```

```
1050 PRINT F$;
1060 NEXT J
1070 END
```

ERASE A LINE OF INFORMATION

To clear a line, you could type

```
10 VTAB 5: REM VERTICAL LINE TO ERASE
20 FOR L = 1 TO 39
30 PRINT " "; REM ONE SPACE BETWEEN QUOTES
40 NEXT L
50 PRINT " "
60 END
```

This does the job, but an easier way is

```
10 VTAB 5: REM VERTICAL LINE TO ERASE
20 CALL -868
30 END
```

We discussed CALL −868 in Chapter 3.

ERASE PORTION OF SCREEN ONLY

To clear only part of the screen, for example, from VTAB 11 to VTAB 17, you could type

```
70 FOR L = 11 TO 17: VTAB L: CALL -868: NEXT L
80 END
```

We can even use this approach to erase the entire screen, as follows:

ERASE ENTIRE SCREEN

"The Invisible Line"

```
10 FOR L = 1 TO 24
20 VTAB L: CALL -868
25 FOR J = 1 TO 50: NEXT J: REM DELAY LOOP
30 NEXT L
40 END
```

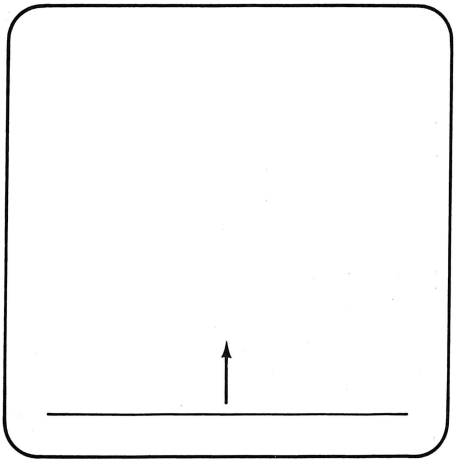


Figure 4-1 Bottoms Up.

"Bottoms Up!"

(Erase screen from bottom up)

```
10 FOR L = 24 TO 1 STEP -1
20 VTAB L: CALL -868
25 FOR J = 1 TO 50: NEXT J: REM DELAY LOOP
30 NEXT L
40 END
```

Now let's add a visible erase line.

"The Blackboard Eraser"

```
10 FOR L = 1 TO 24
20 INVERSE: VTAB L
30 PRINT SPC (40): NORMAL
40 VTAB L: CALL -868
50 NEXT L
60 END
```

We can make the visible line move in reverse (from bottom to top) as follows:

```
10 FOR L = 24 TO 1 STEP -1
20 INVERSE: VTAB L
30 PRINT SPC (40): NORMAL
40 VTAB L: CALL -868
50 NEXT L
60 END
```

The Blackboard Eraser demonstrates a partial screen erase as follows:

```
 10 FOR L = 1 TO 10
 20 INVERSE: VTAB L
 30 PRINT SPC (40): NORMAL
 40 VTAB L: CALL -868
 50 NEXT L
 60 FOR L = 10 TO 1 STEP -1
 70 INVERSE: VTAB L
 80 PRINT SPC (40): NORMAL
 90 VTAB L: CALL -868
100 NEXT L
110 END
```

Here, the eraser deletes just the first ten lines, then returns to the top. By changing the numbers in lines 10 and 60, you can erase any number of lines you desire.

SPECIAL EFFECT SCREEN CLEARS

The following routines erase the screen from different points and in different directions. For clarity, many commands occupy a single line. You may want to use colons to combine several statements on one line.

Note that if you use these routines as subroutines, you must put a RETURN statement at the end.

Split the Seams! (Erase Screen From Middle Out)

```
 50 FOR L = 13 TO 24
 60 HTAB 1: VTAB L: CALL -868
 70 VTAB 25 - L: CALL -868:
 80 FOR J = 1 TO 50: NEXT J
 90 NEXT L
100 END
```

We get this effect by controlling the position of the cursor with the use of HTAB and VTAB (see Chapter 3 for a full explanation). Line 60 uses CALL -868 to erase the first line, beginning at VTAB 13 (the first value of L). Line 70 erases the second line at VTAB 12 (VTAB 25 - L; that is, 25 - 13 = 12).

CALL -868 erases so quickly that the effect would be lost if we did not slow it down. One way to slow programs down is with the SPEED command (see Chapter 6 for a full explanation of its uses).

The only problem here is that SPEED won't work with CALL -868. Instead, we can use a delay loop,

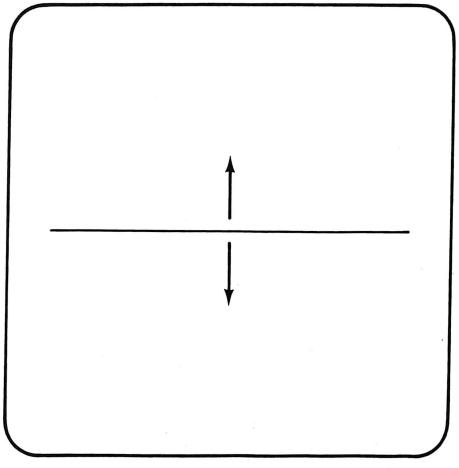


Figure 4-2 Split the Seams

```
FOR J = 1 TO 50: NEXT J
```

This is often referred to as a "do-nothing" loop because it does nothing but take up time.

After the FOR NEXT loop has been executed, NEXT L restarts the erase process with the next two lines, and so on.

Pull That Curtain Aside! (Erase Screen From Right to Left)

```
810 FOR L = 38 TO 0 STEP -1
820 POKE 32, L: POKE 33,40 - L
830 HOME
840 NEXT L
850 END
```

This special effect is accomplished by manipulating the left margin (POKE 32,L) and the width (POKE 33,W) of the text window (see Chapter 3 for full explanation).

We begin by moving the left margin to the right side of the screen. Then, using FOR L = 38 TO 0 STEP −1, we bring the left margin back to its normal position, clearing the screen as it returns.

A Variation of Right to Left

To erase just the lower one-third of the screen right to left, add the following line to the previous routine:

```
800 POKE 34, T
```

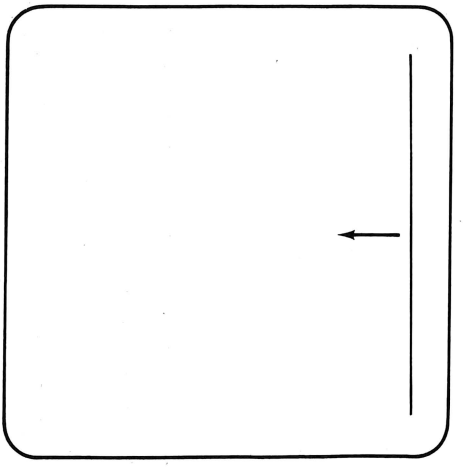


Figure 4-3(a) Pull that Curtain Aside.

where T is the vertical TAB where the erasure begins. For example,

```
800 POKE 34,15
```

erases the bottom part of the screen, from rows 15 through 24.

Don't forget to reset the top margin to the top of the screen. Type

```
900 POKE 34,O
```

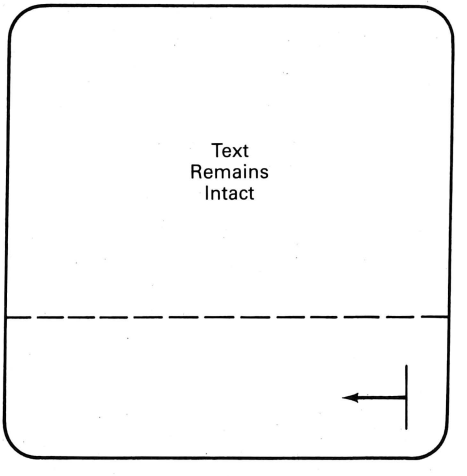


Figure 4-3(b) Variation of Curtain Pull.

Mixed Doubles

Here's a routine that combines together several of the wipes we have learned. Note how the wipes were rewritten to affect only a certain portion of the screen.

```
 10 GOSUB 1000: REM FILL ROUTINE
 20 :
 30 GOSUB 400: REM PAUSE
 40 :
 50 :
 60 :
 70 REM ERASE TOP LEFTHAND QUARTER
 80 :
 90 POKE 33,20: POKE 35,11
100 FOR L = 11 TO 1 STEP -1: VTAB L: CALL -868
110 FOR J = 1 TO 55: NEXT J: NEXT L: VTAB 1
120 :
130 GOSUB 400
140 :
150 REM ERASE BOTTOM RIGHTHAND QUARTER
160 :
170 POKE 35,24: POKE 34,11: POKE 32,20
180 FOR L = 18 TO 23: HTAB 1: VTAB L: CALL -868
190 VTAB 35 - L: CALL -868: FOR J = 1 TO 50: NEXT J: NEXT L
200 :
210 GOSUB 400
220 :
230 REM ERASE UPPER RIGHTHAND QUARTER
240 :
250 POKE 34,0: POKE 35,11
260 FOR L = 1 TO 11: VTAB L: CALL -868: FOR J = 1 TO 55: NEXT J: NEXT L
270 :
280 GOSUB 400
290 :
300 REM ERASE BALANCE OF SCREEN
310 :
320 TEXT: FOR L = 20 TO 0 STEP -1
330 POKE 32,L: POKE 33,40 - L
340 HOME
350 NEXT L
360 :
370 END
380 :
390 :
400 FOR X = 1 TO 500: NEXT X: REM PAUSE
410 RETURN
420 :
```

```
 430 :
1000 REM FILL ROUTINE GOES HERE
1010 :
```

Exit, Stage Right (Left to Right Wipe)

```
10 REM ERASE SUB LEFT TO RIGHT
20 FOR J = 1 to 40
30 POKE 33,J
40 HOME
50 NEXT J
60 END
```

This clear operates on only the width margin. Since normal width is POKE 33,40, the normal margin is restored when the routine finishes.

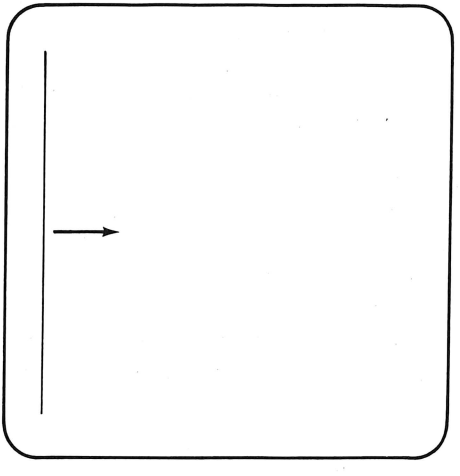


Figure 4-4 Exit Stage Right!

Pull It Over to the Corner, Will Ya'! (Diagonal Screen Erase)

```
10 REM DIAGONAL ERASE BOTTOM RIGHT TO TOP LEFT
   :
   :
50 FOR J = 1 TO 500:NEXT J: REM PAUSE
60 T = 23: L = 39
70 W = 1
80 IF T < = 1 THEN 110
```

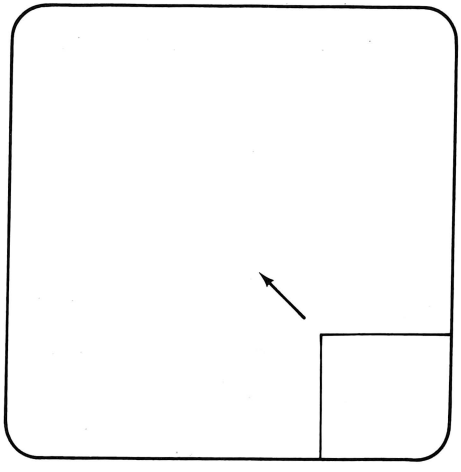


Figure 4-5 Pull It Over.

```
 90 T = T - 1
100 POKE 34, T
110 IF L < = 2 THEN 190
120 L = L - 2
130 W = W + 2
140 POKE 33, W
150 POKE 32, L
160 FOR J = 1 TO 50: NEXT J: REM PAUSE
170 HOME
180 GOTO 80
190 TEXT: REM SETS TEXT WINDOW TO NORMAL VALUES
200 HOME: REM CLEARS ANY REMAINING INFO ON SCREEN
210 END
```

This clever special effect involves manipulating three margins: top, left, and width.

Line 50 is a time delay loop to leave the fill routine on the screen briefly.

Lines 60–70 set new margin parameters.

Lines 80–150 set new margins.

Line 160 slows down the erase routine for observation.

Line 170 clears part of screen.

Line 180 returns to erase the next margins.

Lines 190–200 set the text window to its normal values and clear any remaining information.

Note that in lines 100, 140, and 150 we changed the text window. When this routine is done, we want to reset the text window to its normal values. We could have done this by POKEing in the normal values into line 190; that is,

```
190  POKE 33,40: POKE 32,0: POKE 34,0
```

However, you will notice that we accomplished the same thing and saved a lot of typing by using

```
190  TEXT
```

TEXT automatically sets the text window to its normal values.

You may wonder about the purpose of lines 80 and 110. There are some important restrictions concerning POKEs that affect text window margins. See Chapter 3 for more information.

Not That Corner, The Other One! (Diagonal Erase, Number 2)

This effect is similar to the last diagonal screen erasure. However, since the left margin remains intact, you need only manipulate the width (POKE 33,W) and the bottom margin (POKE 35,B). For a full explanation of line 100, see Chapter 3, "Text Window Manipulation."

```
10  REM DIAGONAL ERASE TOP TO BOTTOM
    :
    :
50  FOR J = 1 TO 500: NEXT J: REM PAUSE
60  B = 1: W = 1
70  IF B = 24 THEN GOTO 100
80  B = B + 1
90  POKE 35,B
```

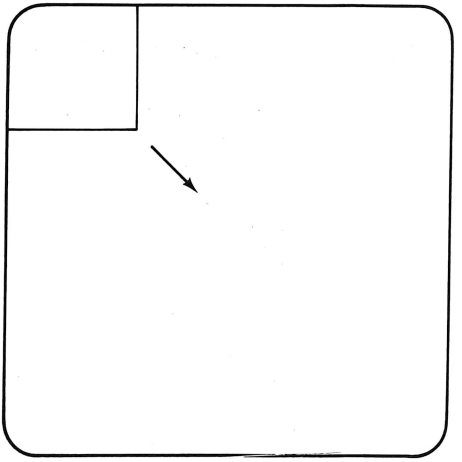


Figure 4-6 Not That Corner, The Other One!

```
100 IF W = >40 THEN 160
110 W = W + 2
120 POKE 33,W
130 FOR J = 1 TO 50: NEXT J: REM PAUSE
140 HOME
150 GOTO 70
160 TEXT
170 HOME
180 END
```

Journey to the Center of the Screen (Outside In)

This routine involves manipulating all four margins to erase the screen from its outer boundaries toward the center.

```
 10 REM FILL SCREEN
 20 FOR J = 1 TO 500: NEXT J: REM TIME DELAY
100 FOR J = 1 TO 12
110 POKE 35,J: CALL -936
120 POKE 34, (24-J): CALL -936
130 POKE 35,24: POKE 34,0
135 IF J = 1 THEN POKE 33,2: GOTO 145
140 POKE 33,J
145 POKE 32, (J - 1)
150 CALL -936: POKE 32, (40 - (2 * J))
160 CALL -936: POKE 32,0: POKE 33,40
170 FOR K = 1 TO 25: NEXT K
180 NEXT J
190 HOME
200 END
```

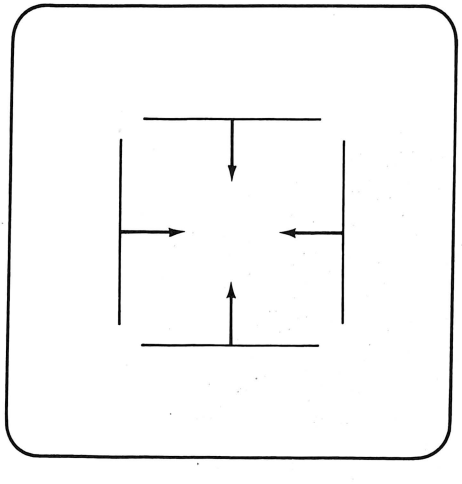
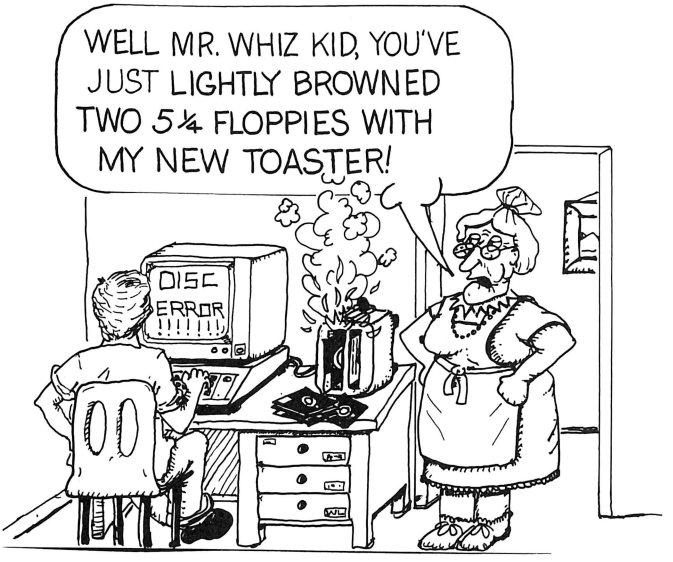


Figure 4-7 Journey to the Center of the Screen.



Line 100 brings the erase routine toward the center.

Line 110 changes the bottom margin. On first pass through the FOR NEXT loop, the bottom margin moves to the top of the screen. CALL −936 (HOME) erase the first line.

Line 120 moves the top margin to the bottom of the screen, POKE 34, (**24 − J**) or POKE 34,**23,** and CALL −936 erases the bottom line.

Line 130 resets both the bottom and top margins (POKE 35,0 and POKE 34,0, respectively) to normal.

Line 135 is used only on the first run through of the FOR NEXT loop to eliminate the two rightmost columns.

Line 140 changes the width, POKE 33,**J** (or 33,**1**) and line 145 the left margin POKE 32,(**J − 1**) or (32,**0**) to make the text window only one column wide (on the first time through the FOR NEXT loop) on the left-hand side.

Line 150 erases the first column with CALL −936. Then POKE 32,((**40 − (2 * J**)) or POKE 32,**38** moves the right side of the screen.

Line 160 erases columns 39 and 40. This completes the first set of erasures. POKE 32,0 and POKE 33,40 reset the left and width margins to normal.

Line 170 is a short delay loop to slow down the erase routine.

Line 180 restarts the erasure.

Line 190 erases the screen after the routine makes its last pass through the FOR NEXT loop.

Close the Curtains! (Erase From Sides Inward)

The performance is over and the audience stands for an ovation as the curtain closes.

The key to making this effect work is POKE 33,W and POKE 32,L.

```
 10 REM ERASE FROM SIDES IN
 20 REM USE FILL ROUTINE HERE
 30 FOR P = 1 TO 1000: NEXT P: REM TIME DELAY
 35 W = 1: L = 39
 40 FOR J = 1 TO 20
 60 POKE 33,W: HOME
 80 POKE 32,L
100 HOME
105 POKE 32,0
110 W = W + 1: L = L - 1
120 NEXT J
130 TEXT
140 END
```

Line 35 sets the right and left margins.

Line 40 makes each margin move toward the center (column 20).

Lines 60–80 POKE in the new margins.

Line 100 clears the screen.

Before reading further, can you see why line 105 is necessary?

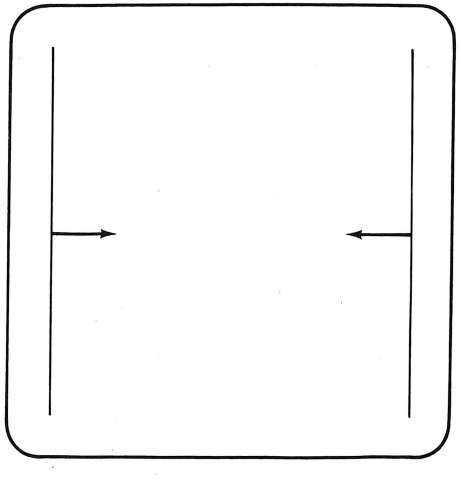


Figure 4-8 Close the Curtains.

Line 105 resets the left margin to normal. Try running the program without it. What happens?

Line 60 changes the width and then executes a HOME command. The first time through the FOR NEXT loop, this clears the screen from the left margin (which, at this point, is still 0) to the width (1).

Line 80 then moves the left margin to column 39, and the HOME command in line 100 erases from there to the right border.

Now line 105 moves back to column 0. The next iteration of line 60 will therefore erase from the left margin (0) to the new width (2). The program thus erases from both edges toward the middle, leaving an even-smaller central section visible.

There is one more reason for line 105. See Chapter 3, "Text Window Massacre," for an explanation of what to watch out for when changing left margin and width.

Line 110 increases the width by one and decreases the left margin by one.

Line 130 resets all margins to normal.

Inverted T Erase

This routine is similar to the previous one. The main difference is in line 112. It puts the bottom margin at the top of the screen, then back to the bottom. This keeps the entire screen from being erased, as in "Close the Curtain!" Just a slight change in this program creates a completely different effect.

```
10  REM INVERTED T ERASE
    :
```

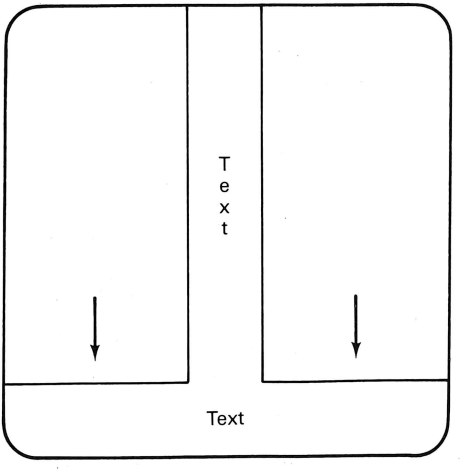


Figure 4-9 Inverted T Erase.

```
 30 FOR J = 1 TO 1000: NEXT J
 32 B = 2
 35 W = 1: L = 39
 40 FOR K = 1 TO 30
 60 POKE 33,W: HOME
 80 POKE 32,L
100 HOME
105 POKE 32,0
110 W = W + 1: L = L - 1
112 POKE 35,B: HOME
116 B= B +1
120 NEXT K
130 TEXT
140 END
```

Close the Window (Erase From Top and Bottom to Middle)

CALL −868 is the key to this effect. A complete line is erased by placing it at the HTAB 1 position of each VTAB position.

```
10 REM ERASE FROM TOP AND BOTTOM TO MIDDLE
  :
  :
50 FOR J = 1 TO 1000: NEXT J: REM PAUSE
60 T = 23
70 FOR L = 24 TO 13 STEP -1
80 VTAB L: HTAB 1: CALL -868
```

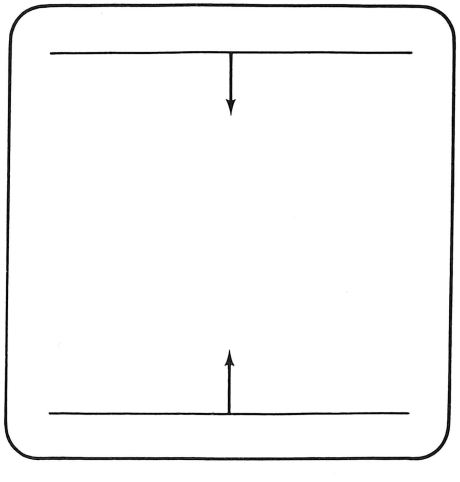


Figure 4-10 Close the Windows.

```
 90 VTAB L - T: CALL -868: FOR J = 1 TO 50:
    NEXT J: T = T - 2
100 NEXT L
110 VTAB 1
120 END
```

Line 70 contains a FOR NEXT loop that counts backward one step at a time. Line 80 begins the erasure from the bottom up, while line 90 begins it from the top down. The delay loop (FOR J = 1 TO 50: NEXT) slows down the erasure. T = T - 2 controls the top margin.

Line 110 leaves the cursor at the top of the screen.

The Camera Lens (Erasure Inside Out)

This routine manipulates all four margins to erase from the inside out. See Figure 4-11. We must be careful to follow the rules governing margin changes.

```
 10 REM ERASE INSIDE OUT
 20 SP = 200: REM CHANGING THIS VALUE (1-255) WILL ALTER SPEED
    :
    :
 60 FOR J = 1 TO 1000: NEXT J: REM PAUSE
 70 T = 11: B = 13: W = 4: L = 18
 80 FOR K = 1 TO 15
 90 POKE 34,T: REM TOP
100 POKE 35,B: REM BOTTOM
```

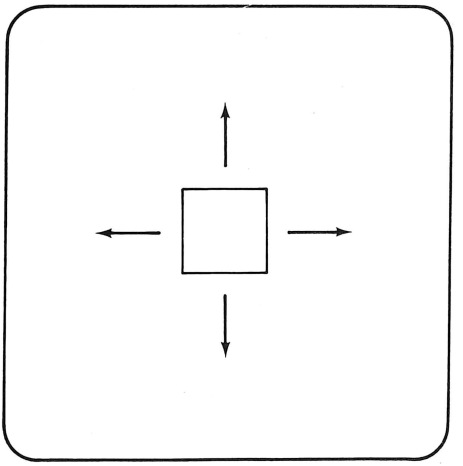


Figure 4-11 The Camera Lens.

```
110 POKE 33,W: REM WIDTH
120 POKE 32,L: REM LEFT
130 HOME
140 SPEED = SP
150 PRINT " "
160 IF T = 0 THEN 180
170 T = T - 1
180 IF B = 24 THEN 200
190 B = B + 1
200 IF W = 40 THEN 220
210 W = W + 4
220 IF L = 0 THEN 240
230 L = L - 2
240 NEXT K
250 SPEED = 255
260 TEXT
270 END
```

Note the **SPEED** command in line 20. We stated previously that POKEs aren't affected by SPEED.

However, if you change the value in line 20 to something between 1 and 255, it affects the speed of the erasure. Before reading further, can you discover how this is possible?

The solution is in line 150. PRINT " " overwrites the screen with spaces. This is the part SPEED controls here.

See if you can figure out why we need lines 160, 180, 200, and 220. (*Hint:* See Chapter 3 on rules for margins.)

The Noisy Camera Lens (Erase Inside Out With Tone)

This is the same as the previous routine, but with an added touch: the Random Tone Generator. As the erasure proceeds, random musical tones are generated. See Figure 4-12.

```
 10 REM ERASE INSIDE OUT WITH TONE
 20 SP = 200: REM CHANGING THIS VALUE (1-255) WILL ALTER SPEED
 30 GOSUB 280
    :
 60 FOR J = 1 TO 1000: NEXT J: REM PAUSE
 70 T = 11: B = 13: W = 4: L = 18
 80 FOR K = 1 to 15
 90 POKE 34,T: REM TOP
100 POKE 34,B: REM BOTTOM
110 POKE 33,W: REM WIDTH
```

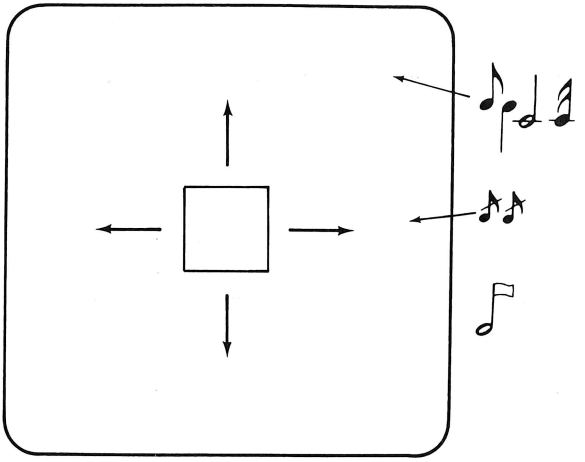


Figure 4-12 The Noisy Camera Lens.

```
120  POKE 32,L: REM LEFT
130  HOME
140  SPEED = SP
150  PRINT " "
154  S = INT ( RND (1) * 200 ) + 10
158  POKE 0,S: POKE 1,10: CALL 771
160  IF T = 0 THEN 180
170  T = T - 1
180  IF B = 24 THEN 200
190  B = B + 1
200  IF W = 40 THEN 220
210  W = W + 4
220  IF L = 0 THEN 240
230  L = L - 2
240  NEXT K
250  SPEED = 255
260  TEXT
270  END
280  FOR J = 771 TO 789: READ X: POKE J,X: NEXT J
290  DATA 173,48,192,136,208,4,198,1,240,8,202,
     208,246,166,0,76,3,3,96
300  RETURN
```

Scroll Up

An eye-catching screen clear scrolls the screen's contents up and out of sight before displaying anything else. We can do this with

```
350  FOR L = 1 TO 24: VTAB 24: PRINT: NEXT L
```

To slow down the scrolling effect, add a delay loop:

```
350  FOR L = 1 TO 24: VTAB 24: PRINT: FOR J = 1 TO 50: NEXT J: NEXT L
```

Try adding this line for an interesting variation:

```
350  FOR L = 1 TO 24: VTAB 24: PRINT: S = PEEK (-16336): NEXT L
```

Incorporating routines in this chapter with those in Chapter 5 will give your programs special qualities that most other programs lack.

5 Screens Alive!

INTRODUCTION

This chapter contains a variety of techniques to manipulate what appears on the screen. Many of the routines contain sound effects (see Chapter 1).

Use these techniques with discretion. An overdose can produce side effects of boredom or irritability.

TYPEWRITER SOUND

To make your Apple sound like a typewriter as it prints, use the following routine:

```
10 HOME
20 S$ = "ATTENTION! THE FOLLOWING MESSAGE": GOSUB 5010
30 S$ = "IS SECRET AND INVISIBLE!!!": GOSUB 5010
35 S$ = "                                ":GOSUB 5010
40 :
```

```
 50 REM REST OF PROGRAM
 60 END

5000 REM SUB FOR TYPEWRITER SOUND
5010 FOR X = 1 to LEN (S$)
5020 SPEED = 125
5030 IF MID$ (S$,X,1) = " " THEN PRINT " ";: GOTO 5100
5040 PRINT MID$ (S$,X,1);
5050 FOR J = 1 TO 2: S = PEEK (-16336): NEXT J
5100 NEXT X
5110 PRINT
5120 SPEED = 255
5130 RETURN
```

WRITING A STRING BACKWARDS

Another interesting effect is making the computer print backwards. This does not mean that the letters are in reverse order. Rather, the word order is reversed, printing everything right to left, last word first.

Sound confusing? Type this routine and it all becomes clear.

Simple Version

```
1000 REM BACKWARDS WRITING SUB
1010 S$ = "FOR SOME REASON, I ALWAYS GET BEHIND."
1020 FOR J = LEN (S$) to 1 STEP -1: VTAB 23: HTAB J:
     PRINT MID$ (S$.J.1): NEXT J
```

We can combine this with the typewriter subroutine.

Typewriter

```
  10 HOME
  20 S$ = "FOR SOME REASON, I ALWAYS GET BEHIND."
  30 GOSUB 1005
  40 END

1000 REM SUB FOR BACKWARDS WRITING
1005 SPEED = 175
1010 FOR X = LEN (S$) TO 1 STEP - 1
1020 VTAB 23: HTAB X
1030 IF MID$ (S$,X,1) = " " THEN PRINT " ": GOTO 1060
1040 PRINT MID$ (S$,X,1)
1050 FOR J = 1 TO 2: S = PEEK (-16336): NEXT J
1060 NEXT X
1070 SPEED = 255
1080 RETURN
```

NOISY, BACKWARDS BILLBOARD

Another version of backwards writing is the billboard routine. This routine in INVERSE, coupled with a sound routine, makes a real eye-catcher.

```
10 REM BACKWARDS BILLBOARD
20 TEXT : HOME : INVERSE
30 M$ = "YOUR APPLE COMPUTER SAYS, GIDDYAP. GO."
40 VTAB 15
50 FOR K = 1 to 40 - LEN (M$) STEP 2:M$ = CHR$ (32) +
   M$ + CHR$ (32): NEXT K
60 FOR J = 39 TO 1 STEP - 1
70 HTAB J: PRINT LEFT$ (M$,40 - J);
80 FOR S = 1 TO 4:X = PEEK ( - 16336): NEXT S
90 FOR K = 1 TO 30: NEXT K: REM PAUSE
100 NEXT J
110 FOR PAUSE = 1 TO 1500: NEXT PAUSE
120 NORMAL : HOME
130 END
```

Source: Public Domain Library.

TITLE WAVE

Here's a catchy way to display your title that's sure to make a splash! See Figure 5-1.

```
10 HOME
20 FOR K = 1 TO 30 STEP 2: HTAB K: PRINT "DIVE": FOR
   J = 1 TO 300: NEXT J: NEXT K
```

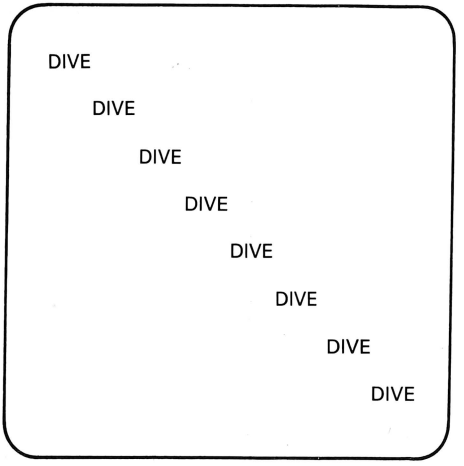


Figure 5-1 Title Wave Demo.

FLASHY ROUTINES

Two common ways to draw attention to a word or phrase are with **INVERSE** or **FLASH.** For example,

```
30 FLASH: PRINT "THIS IS REALLY A FLASH IN THE PAN!": NORMAL
```

FLASH causes the entire sentence to flash. Now substitute INVERSE for FLASH. The message appears in black on white rather than the normal white on black.

You can also draw attention to a particular word by FLASHing or INVERSE-ing it. The following example demonstrates how:

```
30 PRINT "THE LIGHTNING ";:FLASH: PRINT "FLASHED";: NORMAL: PRINT
   " ALL AROUND THEM!"
```

Note that the extra space was added after "THE LIGHTNING . . ." and before ". . . ALL AROUND THEM." This allows for proper spacing in the sentence. The spaces could appear within the quotation marks of "FLASHED" instead (for example, " FLASHED ").

You may have noticed that FLASH makes the entire word or phrase alternate between inverse and normal printing.

What if you just want the word to blink without switching to inverse? The following example shows how:

```
 5. HOME
 6 VTAB 9: PRINT "AROUND THE LAST CURVE . . ."
10 PRINT "DOWN HOME STRETCH . . ."
12 PRINT "THE WINNER IS . . .FLEETFEET"
20 VTAB 11: HTAB 17: PRINT "FLEETFEET"
30 FOR J = 1 TO 225: NEXT J
40 VTAB 11: HTAB 17: PRINT "          "
50 FOR J = 1 TO 120: NEXT J
55 VTAB 22: PRINT "PRESS ANY KEY TO CONTINUE."
60 IF PEEK (-16384) < 127 THEN 20
65 VTAB 11: HTAB 17: PRINT "FLEETFEET"
70 PRINT: END
```

In this example, "FLEETFEET" flashes. Line 20 assumes that the word "FLEETFEET" is on line VTAB 11 and at HTAB 17. When using this technique, change VTAB and HTAB to the position of the characters that you want to flash.

In line 40, the number of spaces between the quotation marks must match the number of characters (including any spaces) that you want to flash. As you can probably guess, we create the flashing by overwriting "FLEETFEET" with spaces. FOR NEXT loops in lines 30 and 50 tell the computer how long to keep the characters on the screen.

SPECIAL ROUTINES

There are other ways to draw attention to particular characters, words, or phrases. For example, if you want the user to respond to a question by pressing a particular key, you could write the program this way:

```
30 PRINT "IS THIS A COLOR OR B/W MONITOR?"
40 PRINT "ENTER (C/B): ";: INPUT AN$
```

You could also write it this way:

```
30 INPUT "IS THIS A COLOR OR B/W MONITOR? (C/B): "ÅN$
```

or this way:

```
30 PRINT "IS THIS A COLOR OR B/W MONITOR?"
40 PRINT " ENTER (C/B): ";: GET AN$
```

Any of the above will work. All are straightforward and commonly used. But they are nothing special. You could use this instead:

```
10 HOME
30 PRINT "IS THIS A COLOR OR B/W MONITOR? ": FLASH: VTAB
   PEEK (37): HTAB 11: PRINT "C";: HTAB 20: PRINT "B";
40 K = PEEK (-16384): IF K < > 194 AND K < > 195 THEN 40
50 POKE -16368,0
60 NORMAL
70 IF K = 194 THEN GOTO (line #) : REM B KEY PRESSED
80 IF K = 195 THEN GOTO (line #) : REM C KEY PRESSED
90 :
100 :
```

Line 30 presents the prompt. VTAB PEEK(37) keeps the cursor on the same line. The letters C and B at HTAB 11 and 20, respectively, flash, giving the user a clue as to what keys are to be pressed.

In line 40 the computer reads the keyboard until the user presses the correct key; that is, C or B.

The following routine uses a special cursor to emphasize material.

```
10 HOME
20 :
80 VTAB 22
90 PRINT "ONCE YOU GET THE HANG OF IT"
100 SPEED = 210
110 FOR J = 1 TO 4: FOR K = 21 TO 18 STEP - 1: VTAB 23:
```

```
        HTAB K: PRINT " ^": NEXT K: FOR K = 18 TO 21: VTAB 23:
        HTAB K: PRINT "  ";: NEXT K: NEXT J
120 SPEED = 255
```

The special cursor looks like the letter V inverted. This character is the shift-N key. In line 110 the **4** determines how many times the special cursor moves back and forth under the word or phrase. The numbers **21** and **18** are the HTAB of the first and last letters in the characters to be underlined. The SPEED, of course, adjusts how fast the special cursor moves. Changing any of the variables affects speed, HTAB, and the number of times that the cursor goes back and forth.

Underlining a Word or Phrase

Another way to draw attention to something is to underline it. The following example demonstrates the routine:

```
10 HOME
20 VTAB 2
30 PRINT "DON'T FORGET TO FEED THE CAT."
40 HTAB 17: PRINT "----"
50 PRINT "I TOLD HIM TO EXPECT YOU."
60 END
```

NOT SO FLASHY ROUTINES

Does the flashing cursor annoy you? When a program requires user input, you may add the following routine.[1] The cursor will either stay on or become invisible, depending on the value of Q (see line 1013).

Unfortunately, this routine disables the arrow keys, so you cannot use them for editing. This may annoy you more than the flashing cursor.

```
  10 REM NOT SO FLASHY
  20 HOME
  30 GOSUB 1000
  40 REM REST OF PROGRAM
  50 END
1000 POKE -16368,0
1005 :
1009 REM ::::::::::::::::::::::::::::::::::::::::::
1010 POKE PEEK (40) + PEEK (41) * Q + PEEK (36),32
1012 :
1013 REM SET Q = 156 OR Q = 256
1014 ::::::::::::::::::::::::::::::::::::::::::::::::
```

[1]Andrew Cutler, "Open Discussion Column," *SOFTALK*, July 1983, p. 52.

```
1020 P = PEEK (-16384): IF P = 141 THEN RETURN:
     REM USER PRESSED 'RETURN'
1030 IF P < = 155 THEN 1010
1040 PRINT CHR$ (P);
1050 POKE -16368,0
1060 X$ = X$ + CHR$ (P): REM X$ = STRING
1070 GOTO 1010
```

MOVING A WORD OR PHRASE ACROSS THE SCREEN

Moving a String Left to Right

```
30 HOME: FOR K = 1 to 14: VTAB 4: CALL -958:
   HTAB K: FOR J = 1 TO 2: S = PEEK (-16336): NEXT J:
   PRINT "SHOW TIME": FOR J = 1 TO 10: NEXT J:
   : NORMAL: NEXT K
40 FOR J = 1 to 700: NEXT J: VTAB 3: HTAB 14: PRINT
   "SHOW TIME"
50 FOR K = 1 TO 14: VTAB 4: CALL -958
   FOR J = 1 TO 2: S = PEEK (-16336): NEXT J: HTAB K:
   PRINT "---- ----": FOR J = 1 TO 10: NEXT J:
   NORMAL: NEXT K
70 :
80 PRINT:PRINT
90 END
```

Line 30 moves the string across the screen, clicking the speaker as it goes. Line 40 moves it up one VTAB. Line 50 moves the underline across VTAB 4, underneath "SHOW TIME."

Moving a String Right to Left

To reverse the direction, type

```
10 HOME
20 S$ = "HERE WE GO . . ."
30 SPEED = 210
40 FOR K = 40 - LEN (S$) TO 1 STEP - 1
45 VTAB 22
46 HTAB K
48 FOR J = 1 TO 2: S = PEEK (-16336): NEXT J
50 PRINT S$;
55 CALL -958
60 NEXT K
70 SPEED = 255
```

Moving a String Up and Down the Screen

This program[2] lets the user pick the direction of the string's movement. There are three choices:

1. Upward. When the string reaches the top, it disappears, and then reappears at the bottom and moves up again.
2. Downward. When the string reaches the bottom, it disappears, and then reappears at the top and moves down again.
3. Bounce Up and Down. The modification suggested in line 100 makes the string move down, then up, and back down again, continually.

Lines 50 through 66 give the user the option of changing the string's direction immediately by pressing the "U" or "D" key.

```
  5 REM VERTICAL MOVING SIGN
    BY NEIL BRONSTEIN
    CALL -APPLE * FEB 1981
 10 HOME: VTAB 8: PRINT "TYPE YOUR VERTICAL MOVING SIGN":
    PRINT :INPUT A$
 15 VTAB 14: PRINT "SHOULD THE SIGN MOVE UP OR DOWN?"
 20 PRINT : GET D$: HOME: VT = 12: X = 1
 25 IF D$ = "U" THEN X = -1
 30 L = (20 - LEN (A$) / 2)
 35 HOME
 40 VTAB VT: PRINT TAB (L) A$;
 50 P = PEEK (-16384)
 60 IF P = 68 THEN X = 1
 65 IF P = 85 THEN X = -1
 66 POKE -16368,0
 70 IF VT = 12 and X = 1 THEN FOR N = 1 TO 1000: NEXT N
 75 VT = VT + X: IF VT > 24 THEN VT = 1
 80 IF VT < 1 THEN VT = 24
 85 GOTO 35
 90 REM PRESS U TO MOVE SIGN UP
        PRESS D TO MOVE SIGN DOWN
100 REM TO BOUNCE THE SIGN ENTER
        75 VT = VT + X: IF VT = 24 THEN X = -1
        80 IF VT = 1 THEN X = 1
```

Coming Together

In this program[3] two lines move toward each other but never quite get together! We can solve the problem by controlling the direction of the second line.

[2]Neil Bronstein, CALL-A.P.P.L.E., February 1981.

[3]Neil Bronstein, CALL-A.P.P.L.E., February 1981.

```
 10 REM HORIZONTAL MOVING SIGN
    BY NEIL BRONSTEIN
    CALL -APPLE * FEB 1981
 20 HOME : VTAB 8: PRINT "TYPE THE FIRST LINE OF YOUR MOVING SIGN"
 30 PRINT : PRINT : INPUT A$:C$ = A$: IF LEN (A$) > 39 THEN 20
 35 LA = 20 - LEN (A$) / 2: FOR N = 1 to LA:A$ = "." + A$: NEXT
 40 FOR N = 1 TO 40 - LEN (A$):A$ = A$ + ".": NEXT
 50 VTAB 14: CALL - 958: PRINT "TYPE THE 2ND LINE (RETURN FOR
    ONE LINE)"
 55 PRINT : INPUT B$: IF LEN (B$) > 39 THEN 50
 60 IF B$ = "" THEN 80
 65 LB = 20 - LEN (B$) / 2: FOR N = 1 to LB:B$ = "." + B$: NEXT
 70 FOR N = 1 TO 40 - LEN (B$):B$ = B$ + ".": NEXT
 75 PRINT : PRINT : PRINT "2ND LINE MOVE LEFT OR RIGHT": PRINT
    : GET E$
 80 HOME
 90 VTAB 8: PRINT A$: VTAB 14: PRINT B$: FOR N = 1 TO 50: NEXT :
    IF MID$ (A$,LA + 1, LEN (C$)) = C$ THEN FOR N = 1 TO 2000:
    NEXT :D = 1
 94 IF D = 0 THEN 100
 95 E = PEEK ( - 16384): IF E = 204 OR E = 210 THEN 100
 97 VTAB 23: PRINT TAB( 6)"PRESS 'L' OR 'R' TO CONTINUE": GOTO 95
100 VTAB 23: CALL - 868:D = 0:A$ = MID$ (A$,2) + LEFT$ (A$,1)
110 IF E = 204 THEN 130
120 B$ = RIGHT$ (B$,1) + LEFT$ (B$,39): GOTO 90: REM
    MOVE RIGHT
130 B$ = MID$ (B$,2) + LEFT$ (B$,1): GOTO 90: REM
    MOVE LEFT
200 REM PRESS 'R' or 'L' DURING
    PROGRAM AND 2ND LINE WILL FOLLOW
    AFTER CENTER PAUSE
```

Lines 20 and 30 request the user to input the first line of information. Line 30 also ensures that the string length does not exceed 39 characters. Lines 35 and 40 redefine the string to include periods before and after the string of characters.

Lines 50 through 75 allow the user to enter a second line of information. Lines 65 and 70 redefine the second string to include periods before and after the string of characters. Line 75 requests the moving direction of the second string.

Line 90 will pause the centered phrase on the screen long enough for the user to easily read it.

Lines 95 through 97 allow the user to change the moving direction of the second line (either right or left) after the center pause. Depending on the user response, the program will branch to either line 120 (moves the second line right) or line 130 (moves the second line left).

Move With the Sound

The following routine combines sound effects with string movement.

```
630 TEXT: HOME
640 FOR S = 768 TO 808: READ P: POKE S,P: NEXT S: FOR S = 815
    TO 837: READ P: POKE: S,P: NEXT S: RESTORE
650 HTAB 12: VTAB 12: PRINT "F R A C T I O N S"
660 FOR S = 15 TO 1 STEP - 1
670 HTAB 12: VTAB 10: PRINT "R E" SPC( S)"D U C I N G ": REM ONE SPACE BEFORE
    QUOTE
680 FOR T = 1 TO 200: NEXT T: CALL 774: NEXT S
690 FOR S = 24 TO 14 STEP - 1: VTAB S: PRINT
    TAB ( 16)"BY" SPC( 40): NEXT S
700 VTAB 14: FOR S = 19 TO 1 STEP - 1: CALL 774: HTAB 17 + S:
    PRINT "BY ";: NEXT S
710 FOR T = 1 TO 200: NEXT T: PRINT
720 PRINT: FOR S = 1 TO 10: VTAB 18: PRINT " " SPC ( S)
    "GARY E. LEONARD": FOR T = 1 TO 10: CALL 774: NEXT T:
    NEXT S
730 INVERSE: VTAB 23: FOR T = 1 TO 40: PRINT " ";: NEXT T : NORMAL
740 FOR T = 1 TO 20: PRINT: CALL 774: FOR S = 1 TO 200:
    NEXT S: NEXT T
750 END

780 DATA 0,0,12,0,0,0,160,0,238,0,3,238,1,3,174,2,3,173,48,
    192,136,208,10, 206,0,3,208,5,206,1,3,240,5,202,240,
    234,208,238,96,0,0
790 DATA 173,48,192,136,208,5,206,46,3,240,9,202,208,245,
    174,45,3,76,47,3, 96,0,0
```

The phrase REDUCING FRACTIONS, lines 650 through 680, is printed to the screen. However, the word REDUCING is split into two parts. RE is printed immediately, while the balance of the word DUCING "clicks" toward the center from the right side. Line 690 prints the word BY quickly scrolling from the bottom. It then darts to the right, up and over underneath the word FRACTIONS, making sounds as it moves. Line 720 similarly prints the author's name before lines 730 and 740 scroll the entire title page up and out of sight.

I'VE BEEN FRAMED!

Inverse Box on Screen

You can use this routine to frame the screen.

```
10 HOME
20 :
```

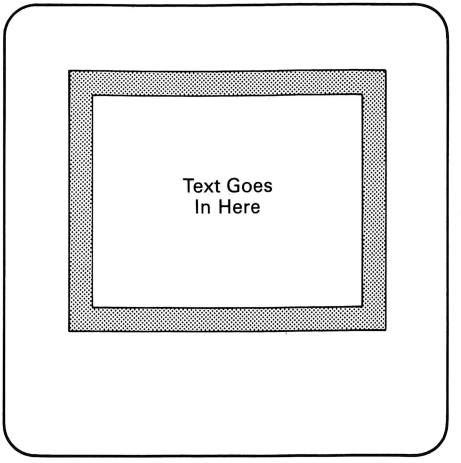


Figure 5-2 I've Been Framed.

```
100 INVERSE: VTAB 1: HTAB 1: FOR J = 1 TO 19:
    VTAB J: HTAB 4: PRINT SPC (37 * (J=1 OR J =
    19)): VTAB J: HTAB 4: PRINT " ";: IF J
    < > 1 AND J < > 19 THEN CALL -868
110 HTAB 40: PRINT " " NEXT J : NORMAL
```

There are two ways to put text inside the frame: (1) Reset the top, bottom, left, and width margins (see Chapter 3 for a full explanation), or (2) use HTAB to indent text.

The following is a sample program using the first option (reset margins):

```
 80 HOME
 90 REM CREATE FRAME
100 INVERSE: VTAB 1: HTAB 1: FOR J = 1 TO 19: VTAB J:
    HTAB 4: PRINT SPC(37 *(J = 1 OR J = 19)): VTAB J:
    HTAB 4: PRINT " ";: IF J < > 1 AND J < > 19 THEN
    CALL -868
110 HTAB 40: PRINT " ": NEXT J
120 NORMAL
200 POKE 34,3: POKE 33,32: POKE 32,6: POKE 35,17:
    REM SET NEW MARGINS
210 HOME
217 VTAB 16: REM SET VTAB LINE TO BEGIN PRINTING
220 S$ = "LADIES AND GENTLEMEN.": GOSUB 1010:
    REM GOES TO TYPEWRITER ROUTINE
225 PRINT:PRINT
230 S$ = "PRESENTING. . . .": GOSUB 1010
240 PRINT:PRINT
250 S$ = "I'VE BEEN FRAMED": HTAB 6:GOSUB 1010
```

```
254 SPEED = 130
260 FOR K = 1 to 9: S = PEEK (-16336): PRINT: NEXT K
265 FOR K = 1 TO 1000: NEXT K
270 S$ = "NEED A LAWYER?!!": HTAB 7: GOSUB 1010
275 SPEED = 130
280 FOR K = 1 TO 7: S = PEEK (-16336): PRINT:NEXT K
290 SPEED = 255
295 VTAB 23
300 FOR K = 1 TO 2000: NEXT K
310 TEXT: HOME: END
320 :
330 :
1000 REM TYPEWRITER SUB
1010 FOR X = 1 TO LEN (S$)
1020 SPEED = 125
1030 IF MID$ (S$,X,1) = " " THEN PRINT " ";: GOTO 1060
1040 PRINT MID$ (S$,X,1);
1050 FOR K = 1 TO 2: S = PEEK (-16336):NEXT K
1060 NEXT X
1070 SPEED = 255
1080 RETURN
```

If you choose the second option (indenting the text), be careful that HTAB and VTAB keep the writing inside the frame.

Thus if you change lines 217 and 220 to

```
217 VTAB 5
220 S$ = "LADIES AND GENTLEMEN.": HTAB 7: GOSUB 1010
```

you must also change the HTABs and VTABs in the remaining PRINT statements.

"Framed Again!"

Here's another way to create an inverse frame.

```
10 HOME
20 INVERSE: VTAB 1: HTAB 1
30 FOR J = 1 TO 39
40 PRINT CHR$ (32);
50 NEXT J
60 FOR J = 1 TO 18
70 VTAB J: HTAB 1
80 PRINT CHR$(32);: HTAB 39: PRINT CHR$ (32);
90 NEXT J
100 VTAB 18: HTAB 1
110 FOR J = 1 TO 39
120 PRINT CHR$(32);
```

```
130 NEXT J
140 NORMAL
150 :
    :
    :
    :
```

Whereas the first frame POKEd new margins, this frame does not. Both examples demonstrate that the programmer can accomplish the same results by different techniques.

Hello Frame

This program presents an INVERSE frame we can use in a HELLO program.

```
 10 CALL -936: FOR I = 1069 TO 1099: POKE I,32:
    POKE I + 896,32: NEXT I
 20 FOR I = 1197 TO 1867 STEP 128: POKE I,32: POKE I + 30,32:
    NEXT I
 30 VTAB 11: HTAB 9: PRINT "MISCELLANEOUS PROGRAMS"
 40 VTAB 13: HTAB 17: PRINT " date goes here"
 50 VTAB 14: HTAB 13: PRINT "MASTER-DOS 3.3"
 90 FOR A = 1 to 2500: NEXT A: REM PAUSE
 95 CALL -936: REM HOME
100 PRINT CHR$(4); "CATALOG"
110 END
```

Source: Public Domain Library.

Star Frame

You can emphasize important information by framing it in stars, as in Figure 5-3. The following routine prints the frame, and then displays the message.

```
  10 HOME
     :
1000 REM STAR FRAME
1010 VTAB 1: FOR J = 1 TO 40: HTAB J: PRINT "*";:NEXT J
1020 FOR J = 1 to 22: HTAB 40: VTAB J: PRINT "*";:NEXT J
1030 VTAB 21: FOR J = 40 TO 1 STEP -1: HTAB J: PRINT "*";:NEXT J
1040 FOR J = 22 to 1 STEP -1: HTAB 1: VTAB J: PRINT "*";: NEXT J
1045 :
1050 REM USE THESE POKES
1060 POKE 33,37: POKE 32,2: POKE 34,2: POKE 35,21 : SPEED = 100
1065 HOME
1070 REM USE FOLLOWING LINES TO WRITE YOUR MESSAGE IN
```

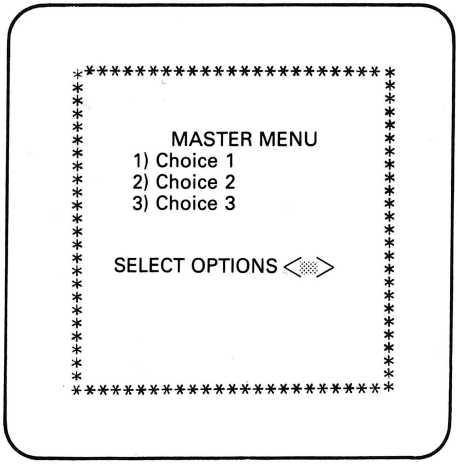


Figure 5-3 Star Frame.

```
1075 :
1080 :
1090 FOR J= 1 TO 25: PRINT: NEXT J
1100 TEXT: SPEED = 255
1110 VTAB 22: END
```

The star frame is excellent for a master menu, message, instructions, etc. Try replacing the asterisk with other characters, such as a colon (:) on the side borders and a period on the top and bottom borders.

SPIN MASTER MENU, elsewhere in this chapter, uses the star frame with other routines.

THE BULLETIN BOARD HORIZONTAL SCROLLING

Another useful technique for displaying information is to scroll it. This creates the effect of an electronic billboard.[4]

```
10 HOME
15 S$ = TEXT SCROLLS ACROSS THE SCREEN. THIS TECHNIQUE
   CAN PRODUCE MESSAGES, INSTRUCTIONS OR
   EVEN.....WARNINGS ! ! !" +
   "                                ": REM 28 SPACES
20 GOSUB 40
25 S$ = "SEE HOW MUCH CAN BE WRITTEN IN SUCH A SMALL SPACE !"
   + "                              ": REM 28 SPACES
```

[4]Published in *inCider Magazine,* Wayne Green Inc., June 1983.

```
 26 GOSUB 40
 30 END
 40 REM N = NUMBER OF TIMES YOU WANT MESSAGE TO REPEAT
 42 N = 1
 45 FOR K = 1 TO N
 46 VTAB 22
 50 H = 1
 70 FOR J = 1 TO LEN (S$)
 80 HTAB 29 - H: PRINT MID$ (S$, J - H + 1,H);
 90 H = H + 1: IF H > 26 THEN H = 26
100 FOR T = 1 TO 50: NEXT T: REM TIME DELAY TO CONTROL
    SPEED OF SCROLL
110 HTAB 2: PRINT SPC (26);
120 NEXT J
130 NEXT K
140 RETURN
```

MAY I QUOTE YOU?

Have you ever wanted to use quotation marks in a sentence? You probably couldn't get the computer to print the quotation marks. Normally, it prints whatever is inside quotation marks, but not the marks themselves. For example, let's say you want the following statement to appear exactly as shown:

```
HE VOTED FOR THE "LAME DUCK" AMENDMENT
```

If you type

```
20 PRINT "HE VOTED FOR THE "LAME DUCK" AMENDMENT"
```

the result will not be what you want. To make the computer print quotation marks, we must use **CHR$ (34).**

```
20 Q$ = CHR$ (34): REM ALLOWS QUOTATION MARKS
30 PRINT "HE VOTED FOR THE " + Q$ + "LAME DUCK" + Q$ +
   " AMENDMENT."
```

CENTERING WORDS OR PHRASES

To center a word or phrase, use the following subroutine.

```
10 S$= "TYPE INFO YOU WANT TO CENTER"
20 GOSUB 1010
30 :
40 REM REST OF PROGRAM
```

```
1000 REM CENTERING SUB
1010 HTAB 20 - LEN (S$)/2
1020 PRINT S$
1030 RETURN
```

If you have an 80-column screen, line 1010 should be

```
1010 HTAB 40 - LEN (S$) / 2
```

An example The high-resolution game *Sherwood Forest* presents the introduction and titles by centering them on the top line of the text window. This technique can add a professional-looking touch and some variety to your program.

```
10 S$ = "BILLING PROGRAM": GOSUB 1010
20 S$ = "WRITTEN BY JOE SMITH": GOSUB 1010
30 S$ = "COPYRIGHT 1882": GOSUB 1010
40 END

1000 REM CENTER ROUTINE
1010 HOME
1020 CALL -198
1030 VTAB 1
1040 CALL -868
1050 HTAB 20 - LEN (S$)/2
1060 PRINT S$
1070 FOR J = 1 to 1000: NEXT J
1080 RETURN
```

Lines 10–30 assign the string variable S$ and send the program to line 1010.

Lines 1000–1080 center the phrase.

Line 1020 produces a tone or beep each time a new title is presented. Try adding a different tone routine for variety (See Chapter 1).

Line 1030 makes the phrase appear at VTAB 1.

Line 1040 clears any previous information on VTAB 1.

Line 1070 creates a short delay for reading the titles or introduction. You can change the FOR NEXT loop to any time you desire.

TITLE MAKER

Here's an exciting and unusual screen display to introduce a program.

```
10 REM **************************
20 :
30 REM     TITLE MAKER
```

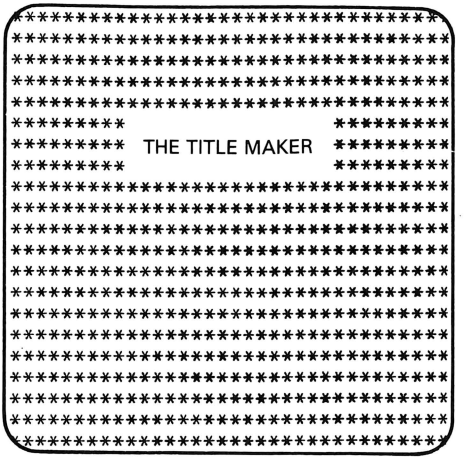


Figure 5-4 Title Maker.

```
 40 :
 50 REM      (C) 1984 WAYNE DYRNESS
 60 :
 70 REM *************************
 80 :
 90 :
100 GOSUB 310: REM ASSIGNS STRING VALUE
110 :
120 GOSUB 480: REM FILL SCREEN
130 :
140 FOR C = 1 to 5
150 GOSUB 390: REM CENTERS STRING
160 VTAB 7: HTAB H: PRINT SP$
170 VTAB 8: HTAB H: PRINT S$(C)
180 VTAB 9: HTAB H: PRINT SP$
190 FOR PAUSE=1 TO 1500: NEXT PAUSE
200 GOSUB 490: REM REFILL SCREEN
210 NEXT C
220 :
230 REM SIDE CLEAR
240 :
250 FOR CLEAR = 1 to 40
260 POKE 33,X: HOME
270 NEXT CLEAR
280 :
290 REM STRING ASSIGNMENT
300 :
310 S$(1) = " THE SOFTWARE COMPANY "
320 S$(2) = " PRESENTS "
330 S$(3) = " THE FABULOUS "
```

```
340 S$(4) = " THE INCREDIBLE "
350 S$(5) = " TITLE MAKER! "
360 :
370 REM CENTERS STRING
380 :
390 H = 20 - LEN (S$(C)) / 2
400 SP$ = " "
410 FOR K = 1 TO LEN (S$(C)) - 1
420 SP$ = SP$ + " "
430 NEXT K
440 RETURN
450 :
460 REM FILL SCREEN ROUTINE
470 :
480 HOME
490 F$ = "****************************************":
    REM 40 ASTERISKS
500 FOR J = 1 TO 23
510 VTAB J
520 HTAB 1
530 PRINT F$;
540 NEXT J
550 RETURN
560 :
570 :
580 REM REST OF PROGRAM
    :
    :
    :
```

Line 100 assigns the string value.

Line 120 fills the screen with asterisks.

Line 140 begins the FOR NEXT loop for five string arrays.

Line 150 centers the string array.

Lines 160–180 print a border of spaces around the phrase.

Line 170 prints the phrase.

Line 190 creates a short delay for reading the phrase.

Line 200 refills the screen with asterisks.

Lines 230–270 clear the screen with a cinematic effect after all phrases have been PRINTed.

Lines 290–350 assign phrases to string arrays. Note the two spaces before and after the phrase within the quotation marks. This is part of the border around the phrase.

Line 390 calculates the centering.

Figure 5-5 Title Wave.

Line 400 establishes the variable SP$ as one space.

Lines 410–430 determine the border of spaces to surround the phrase.

Lines 460–550 fill the screen with asterisks.

Lines 580 on are for the rest of your program.

GETTING THE STRAIGHT ANSWER

There are many ways to request user response in a program. Two of the most common are INPUT and GET.

Another technique, called a **SPIN,** is sure to get the user's attention! What is a SPIN? Type this program and see.

```
 5 REM SPIN

10 A$ = "|/-" + CHR$ (92)
20 HOME: PRINT "PRESS ANY KEY: ";
30 FOR X = 1 TO 4: VTAB 1: HTAB 16: PRINT MID$ (A$,X,1)
40 FOR Y = 1 TO 20: NEXT Y,X
50 KEY = PEEK (49152): IF KEY < 128 THEN 30
60 POKE 49168,0: VTAB 1: HTAB 16: PRINT CHR$ (KEY - 128)
```

Source: "Twirling Cursor," Public Domain Software.

Line 10 creates the SPIN. **CHR$ (92)** is a backslash (\). You cannot type it from the keyboard, but PRINT CHR$(92) prints it on the screen.

Line 30 creates a FOR NEXT loop to print each character in the SPIN.

VTAB 1 and HTAB 16 ensure that each character is printed at the same location. Line 40 is a short delay loop to control the speed of rotation. The illusion is one of a twirling stick.

Line 50 causes the computer to strobe (read) the keyboard until any key is pressed.

POKE 49168,0, in line 60, resets the keyboard strobe so that the next character may be read in and then prints the key typed.

The next program, SPIN MASTER MENU, demonstrates the use of SPIN.

Spin Master Menu

```
 10 REM MASTER MENU WHICH LOADS SELECTED PROGRAMS
 30 :
 40 :
 70 Z$ = "< >" + CHR$(8) + CHR $(8)
 90 D$ = CHR$(4)
110 HOME
130 VTAB 3
150 CALL -868: HTAB 15: INVERSE: PRINT "MASTER MENU": NORMAL
170 PRINT: PRINT
190 HTAB 3
210 PRINT "1 - BILLING"
230 HTAB 3
250 PRINT "2 - ACCOUNTS RECEIVABLE"
270 HTAB 3
290 PRINT "3 - ACCOUNTS PAYABLE"
310 HTAB 3
330 PRINT "4 - SALES REPORT"
430 HTAB 3
450 PRINT "5 - CATALOG OF DISK"
470 HTAB 3
530 PRINT "6 - EXIT PROGRAM"
550 VTAB 20
570 PRINT TAB (18) "COPYRIGHT SEPT, 1992"
590 PRINT TAB (18) "     JOE SMITH"
610 VTAB 1: FOR X = 1 TO 40: PRINT "*";: NEXT X: VTAB 23:
    FOR X = 1 TO 40: PRINT "*";: NEXT X
630 FOR X = 1 TO 22: VTAB X: PRINT "*": NEXT X
650 FOR X = 1 TO 22: HTAB 40: VTAB X: PRINT "*": NEXT X
670 VTAB 16
690 HTAB 3
708 A$ = "!/-" + CHR$ (92)
```

```
710 PRINT "SELECT AN OPTION: ";
718 FOR X = 1 TO 4: VTAB 16: HTAB 19: PRINT Z$: VTAB 16:
    HTAB 20: PRINT MID$(A$,X,1)
720 FOR Y = 1 TO 20: NEXT Y,X
722 KEY = PEEK (49152): IF KEY < 128 THEN 718
724 POKE 49168,0: VTAB 16: HTAB 20: PRINT CHR$ (KEY - 128)
730 LET X$ = CHR$ (KEY - 128): X = ASC (X$) - 48
750 IF X$ = CHR$(13) THEN 718
770 VTAB 18: HTAB 3: PRINT "**** ";: FLASH: PRINT "LOADING PROGRAM";:NORMAL:
    PRINT " ****"
790 IF X = 1 THEN PRINT D$;"RUN BILLING"
810 IF X = 2 THEN PRINT D$;"RUN A/R PROGRAM"
830 IF X = 3 THEN PRINT D$;"RUN ACCT PAY"
850 IF X = 4 THEN PRINT D$;"RUN SALES REPORT"
910 IF X = 5 THEN HOME: CALL -868: HTAB 15: INVERSE:
    PRINT "CATALOG": NORMAL: POKE 34,3: PRINT D$;"CATALOG":
    PRINT: PRINT "PRESS ANY KEY TO CONTINUE.";:GET A$:PRINT:
    POKE 34,0: GOTO 10
950 IF X = 6 THEN PRINT: VTAB 18: HTAB 3: INVERSE:
    PRINT "PROGRAM HAS BEEN EXITED.";:NORMAL:
    CALL -868: CALL -198: CALL -198: END
960 GOTO 718
```

There are several ways in which this program can be written more efficiently. Let's discuss just one of them. A menu-driven program is a perfect opportunity to use POKEs for margin control. Note that the menu options are indented HTAB 3. By eliminating lines 190,230,270,310,430, and 470 and replacing them with

```
190 POKE 32,2: POKE 33,38: REM MOVE LEFT MARGIN
192 PRINT
    :
    :
    :
540 POKE 32,0: POKE 33,40: REM RESET MARGINS
```

the program will function as before, but with fewer lines. Line 190 moves the left margin to the right, so characters begin printing in what used to be HTAB 3 or column 3. Line 540 restores the margin to normal. For more information on POKEs 32 and 33, see Chapter 3.

Spin Writing

SPIN gives us a novel way of displaying information.

```
10 REM ************************
20 REM     SPIN WRITING
```

```
 30 REM (C) 1984 WAYNE DYRNESS
 40 REM ************************
 50 HOME
 60 GOSUB 900
100 HOME
110 S$ = "INTRODUCING . . . A NEW PROGRAM THAT PEOPLE"
120 GOSUB 600
125 PRINT
130 S$ = "HAVE BEEN FLIPPING HEAD OVER HEELS FOR . . ."
140 GOSUB 600
300 END
600 FOR Z = 1 TO LEN (S$)
708 A$ = "!/-" + CHR$(92)
710 FOR X = 1 TO 4: HTAB Z: PRINT MID$ (A$,X,1);
720 FOR Y = 1 TO 10: NEXT Y,X
730 :
800 HTAB Z
810 S = PEEK (-16336) + PEEK (-16336)
830 NEXT Z
840 RETURN
900 FOR T = 1 TO 10
910 FOR X = 1 TO 4: HTAB 1: PRINT MID$ (A$,X,1);
912 S = PEEK (-16336)
920 FOR Y = 1 TO 20: NEXT Y,X
925 NEXT T
930 RETURN
```

Line 900 adds dramatic effect. It gets the SPIN warmed up and ready to take off.

Inverse Field Input Subroutine

The subroutine[5] within the following program calculates and displays the maximum lengths of data items. If the user tries to exceed them, a bell rings.

```
 5 REM INDICATE LENGTH OF ENTRY
10 REM INVERSE FIELD INPUT SUBROUTINE
   BY CHARLES HEARNE
   CALL -A.P.P.L.E. * JUNE 1981
20 GOTO 210: REM BRANCH TO TEST PROGRAM
25 REM
   IS$=INPUT STRING
   IC$=INPUT CHARACTER
   VT=VERT TAB POSN OF FIELD START
   HT=HORIZ TAB POSN OF FIELD START
```

[5]Charles Hearne, CALL-A.P.P.L.E., June 1981.

```
      FL=FIELD LENGTH
      CP=CURSOR POSN IN FIELD
 40   REM
      LINE 50 BEGINS SUBROUTINE TO
      SET INVERSE FIELD, SET INPUT
      STRING TO NULL, TAB TO BEGINNING
      OF FIELD, AND SET CP TO 1
 50   INVERSE: VTAB VT: HTAB HT: IS$ = " ": CP=1
 60   GET IC$: IF IC$ = CHR$ (27) THEN NORMAL: RETURN: REM
      READ CHR: CK FOR ESC KEY
 70   IF IC$ = CHR$ (13) THEN 130: REM
      CK FOR C/R
 80   IF IC$ = CHR$ (8) THEN 150: REM
      CK FOR LEFT ARROW
 90   IF CP > FL THEN PRINT CHR$ (7);: GOTO 60: REM
      CK FOR MAX FIELD LENGTH
100   IF IC$ = CHR$ (21) THEN 180: REM
      CK FOR RIGHT ARROW
110   IF ASC (IC$) < 32 THEN PRINT CHR$ (7);: GOTO 60:
      REM DELETE THIS LINE TO ALLOW CONTROL CHAR TO BE INPUT
120   IS$ = IS$ + IC$: PRINT IC$;:CP = CP + 1: GOTO 60: REM
      ADD LAST CHR TO STRING: PRINT CHR: INCREMENT CP:
      BRANCH BACK FOR NEXT CHR
130   IF CP < = FL THEN FOR K = CP TO FL: PRINT " ";: NEXT: REM
      IF C/R. BLANK REST OF FIELD
140   NORMAL: RETURN
150   CP = CP - 1: IF CP = 0 THEN PRINT CHR$ (7);: GOTO 50: REM
      REMOVE LAST CHR FROM STRING
160   PRINT IC$;: IF CP > 1 THEN IS$ = LEFT$ (IS$,CP - 1): GOTO 60
170   GOTO 50
180   AS = PEEK ( PEEK (40) + 256 * PEEK (41) + PEEK (36)): IF AS
      < 32 THEN IC$ = CHR$ (AS + 64): GOTO 120: REM
      IF RIGHT ARROW. READ INVERSE CHR
      AT CURSOR AND CONVERT TO NORMAL,
      ADDS CHR TO STRING
190   IC$ = CHR$ (AS): GOTO 120

200   REM

***  TEST PRGM TO ILLUSTRATE ***
***  INVERSE FIELD INPUT RTN ***

210   HOME: VTAB 4: HTAB 11: PRINT "NAME: ";:FL = 20: GOSUB 500
220   VTAB 7: HTAB 2: PRINT "TELEPHONE NO.: ";:FL = 12: GOSUB 500
230   VT = 4: HT = 17: FL = 20: GOSUB 50: NA$ = IS$
240   VT = 7: HT = 17: FL = 12: GOSUB 50: PH$ = IS$
250   VTAB 15: HTAB 11: PRINT "NAME:"NA$
```

```
260 VTAB 16: HTAB 2: PRINT "TELEPHONE NO.: "PH$
270 END
500 INVERSE: FOR K = 1 TO FL: PRINT " ";: NEXT: NORMAL: RETURN
```

? Extra Not Ignored

APPLESOFT won't let an INPUT statement contain a comma or colon. It truncates the statement and prints an "?Extra Ignored" message. That is, it ignores anything entered after a colon or comma. The followng subroutines let you type anything in an INPUT statement.

```
  10 REM COMMA/COLON INPUTTER
  20 PRINT "TYPE ANYTHING (COMMAS OK): ";
  30 GOSUB 9000
  40 PRINT A$
  50 END
9000 CALL 64879: A$ = "": FOR N = 512 TO 767: IF PEEK (N)
     < > 141 THEN A$ = A$ + CHR$ (PEEK (N) - 128): NEXT
9020 RETURN
```

How about this one?

```
100 LET IN$ = "X"
110 TEXT : POKE - 16298,0: HOME
120 REM
130 REM THE FIRST VARIABLE
140 REM DEFINED MUST BE A STRING
150 REM THIS STRING WILL REC'VE
160 REM INPUT FROM THE CALL
170 REM
180 REM THIS POKES THE INPUT
190 REM SIMULATOR ROUTINE
200 REM INTO MEMORY . . .
210 REM
220 FOR J = 768 TO 788
230 READ K
240 POKE J,K
250 NEXT J
260 DATA 162,0,32,117,253,160,2
270 DATA 138,145,105,200,169,0
280 DATA 145,105,200,169,2,145
290 DATA 105,96
300 REM
310 REM NOW TO USE IT!
320 REM
330 PRINT "TYPE ANY CHARACTERS YOU WANT:"
340 PRINT
```

```
350 CALL 768:IN$ = MID$ (IN$,1)
360 REM
370 REM THIS IS AN "INPUT IN$"
380 REM BUT IGNORES "," & ":"
390 REM
400 PRINT
410 PRINT "AND HERE'S WHAT YOU TYPED IN:"
420 PRINT : PRINT IN$
430 PRINT
440 PRINT "NOTE THAT EVEN QUOTES, COMMAS AND"
450 PRINT "COLONS GET THROUGH UNSCATHED."
460 PRINT : PRINT "NOW LET'S WRITE IT TO THE DISK."
470 PRINT CHR$ (4);"OPEN LITERAL"
480 PRINT CHR$ (4);"WRITE LITERAL"
490 PRINT IN$
500 PRINT CHR$ (4);"CLOSE"
510 PRINT : PRINT "AND READ IT BACK IN . . ."
520 LET IN$ = " "
530 PRINT CHR$ (4);"OPEN LITERAL"
540 PRINT CHR$ (4);"READ LITERAL"
550 CALL 768:IN$ = MID$ (IN$,1)
560 PRINT CHR$ (4);"CLOSE"
570 PRINT : PRINT IN$
580 PRINT : PRINT "TA-DAA!!": END
```

Source: Public Domain Software.

What's That You Say??

Read the keyboard PEEK (−16384). A cousin of the GET statement is **PEEK (−16384).** When using GET, you usually set a particular key to a variable. For example,

```
20 PRINT "IS THIS OK? (Y/N) ";:GET A$
```

asks for a "yes" or "no" response. A$ will then be equal to whichever key the user presses.

But what if you don't want to set a particular key to a variable? The following routine lets the user "PRESS ANY KEY TO CONTINUE" after executing part of a program.

```
290 VTAB 23: PRINT "PRESS ANY KEY TO CONTINUE."
300 IF PEEK (−16384) < 127 THEN 300
310 POKE −16368,0
320 :
330 REM REST OF PROGRAM
```

Line 300 instructs the computer to read the keyboard. When the user presses a key (the ASCII value of any key is greater than 127), the program continues. If no key is pressed, then line 300 instructs the computer to go back and read the keyboard again until a key is pressed. Line 310 resets the keyboard strobe.

Could You Repeat That, Please?

Peek (49152). **PEEK (49152)** works in the same way as POKE −16368; that is, it makes the computer read the keyboard. This program reads the keyboard and produces the ASCII value plus 128 of the last key pressed before PEEK. To cancel the keypress, use POKE 49168,0.

```
10 PRINT:PRINT "PRESS A KEY."
20 K = PEEK (49152) - 128
30 IF K < 32 THEN GOTO 20
40 POKE 49168,0
50 PRINT "YOU JUST PRESSED THE <";
60 PRINT CHR$(K);"> KEY."
70 END
```

Error-Proof User Input

The following routine instructs the user on how to enter the correct response to an INPUT statement.

```
10 HOME
20 PRINT "IS THIS CORRECT?"
25 PRINT "USE ARROW KEYS <- - -> TO MOVE CURSOR":
   PRINT "TO CORRECT ANSWER. THEN PRESS <RETURN>."
28 PRINT
30 GOSUB 110
40 IF H = 1 THEN GOTO (line #)
50 IF H = 2 THEN GOTO (line #)
60 REM REST OF PROGRAM
99 END
100 REM SUB TO INPUT YES OR NO
110 H = 1: REM DEFAULT VALUE=YES
120 VT = PEEK (37) + 1
130 VTAB VT: INVERSE: PRINT "YES";: NORMAL: PRINT "/NO"
140 GOTO 170
150 VTAB VT: PRINT "YES/";: INVERSE: PRINT "NO": NORMAL
170 KEY = PEEK (49152) - 128
180 IF KEY < > 8 THEN 190: REM LEFT ARROW
185 POKE 49168,0: H = 1: GOTO 130
190 IF KEY < > 21 THEN 200: REM RIGHT ARROW
195 POKE 49168,0: H = 2: GOTO 150
```

```
200 IF KEY < > 13 THEN 170: REM RETURN KEY
210 POKE 49168,0
220 RETURN
```

MENU MAGIC

Here's a menu that uses an automatic scrolling technique. Line 130 turns the scrolling off.

```
 10 GOTO 1000
    :
    :
100 REM ********************
105 REM >> MENU MAGIC <<
106 REM
110 REM BY DR. R. F. ZANT
120 REM ********************
130 C = 32: REM CHANGE TO C=0 IF NO AUTO SCROLL DESIRED AT FIRST
140 L = BEGIN: GOSUB 210
150 FOR J = 0 TO 1: GOSUB 305: J = (C=13)
160 NEXT J
170 L = L - BEGIN + 1
180 RETURN
200 REM >> PRINT "->" ON LINE 'L' <<
210 HTAB 1: VTAB L
220 PRINT "->";
230 RETURN
300 REM >> GET CHAR. MOVE '->' <<
305 IF C = 32 THEN IF PEEK (-16384) < 127 THEN 320
310 WAIT - 16384,128: C = PEEK (-16384) - 128: POKE -16368,0
320 IF C = 32 OR C = 21 THEN GOSUB 410
325 IF C = 32 THEN FOR C = -300 TO 31: NEXT C
330 IF C = 8 THEN GOSUB 510
340 RETURN
400 REM >> MOVE DOWN ONE LINE <<
410 GOSUB 610
420 L = L + 1: IF L > LAST THEN L = BEGIN
430 GOSUB 210
440 RETURN
500 REM >> MOVE UP ONE LINE <<
510 GOSUB 610
520 L = L-1: IF (L < BEGIN) THEN L = LAST
530 GOSUB 210
540 RETURN
600 REM >> CLEAR '->' <<
610 HTAB 1: PRINT "  ";: REM 2 SPACES
```

```
 620 RETURN
1000 REM >> SAME MENU <<
1010 REM << FIRST PRINT MENU ITEMS >>
1020 REM << LEAVE COL. 1 & 2 BLANK >>
1030 HOME: HTAB 10: PRINT "SAME MENU"
1040 VTAB 5: PRINT " FIRST CHOICE": REM INDENT 2 SPACES
1050 PRINT " SECOND CHOICE"
1060 PRINT " THIRD CHOICE"
1070 PRINT " FOURTH CHOICE"
1080 PRINT " FIFTH CHOICE"
1090 VTAB 17: PRINT " PRESS SPACE BAR TO SCROLL POINTER.":
     PRINT "ANOTHER KEY TO STOP SCROLL. ARROW KEY":PRINT "TO
     MOVE POINTER. RETURN KEY TO ENTER":PRINT "CHOICE."
1200 REM >> THEN SET BEGIN AND LAST <<
1210 BEGIN = 5: LAST = 9
1220 REM >> THEN CALL MENU ROUTINE <<
1230 GOSUB 130
1240 VTAB 22: HTAB 1: PRINT "CHOICE IS ";L
1250 END
```

Source: Public Domain Library.

6 Your Error-Oneous Zone is Showing

INTRODUCTION

You slave for hours over a hot keyboard, creating what you think is the ultimate program. Beaming with the pride of a new father or mother, you hand over your newborn creation to another computer user. Anxiously, you wait for the "oohs" and the "ahhs" you're sure are forthcoming as the other person begins to explore the marvels of the "perfect" program.

But wait! Something unexpected happens! "Syntax error" or "End of Data" appears. The screen or the printer goes crazy! You gasp in horror.

"What have you done to my program?!!" you scream. "That's never happened before!"

Does the situation sound familiar? As much as we pride ourselves on our programming skills, problems can occur, or, as Murphy said, "will."

You can often prevent your program from stopping unexpectedly or losing important data if you take likely problems into account. This is called *error trapping*.

In this chapter, we will discuss several ways to trap programming and DOS-related errors and how to avoid or correct them.

STRING FUNCTIONS

This section will review error trapping through the use of string functions. Strings can be made up of letters, numbers, spaces, or symbols.

LEN

LEN is the length function. It is valuable to the programmer who is working with fixed-length entries. For example, in

```
100 INPUT "ENTER NAME (MAX 20 CHAR): ";N$
```

line 100 instructs the user to enter a name, but not to exceed 20 characters. The string variable N$ will contain the user's entry.

We need some way to detect when the user enters more than 20 characters. Enter LEN to the rescue!

```
100 INPUT "ENTER NAME (MAX 20 CHAR): ";N$
110 IF LEN (N$) > 20 THEN CALL -198: PRINT "ENTRY TOO LONG!":
    GOTO 100
```

Line 110 finds any string variable N$ that is greater than 20 characters, then PRINTs an error message and returns to line 100 for another entry.

The format of LEN is

LEN (sexpr)

where sexpr is equal to a string variable.

This function will return the number of characters in the string, between 0 and 255. A string containing more than 255 characters (which sometimes happens if the string variable is a concatenation of strings) produces a ?STRING TOO LONG ERROR.

VAL

VAL is short for VALue. A number within a string variable, e.g., "156", cannot be interpreted arithmetically. In other words, typing

```
10 PRINT "156" + 5
```

displays a ?TYPE MISMATCH ERROR message.

To the computer, a number within quotation marks is considered a character,

just like a letter or a question mark. To make the computer read the numerical VALue of a string's contents, we use the VAL function. Its format is

VAL (sexpr)

where sexpr is equal to a string variable.

Typing

```
10 PRINT VAL ("156") + 5
```

displays the correct answer, 161. We could also use the form

```
10 N$ = "156"
20 PRINT VAL (N$) + 5
```

The use of VAL is important in programming. Let's demonstrate an example of it.

```
100 INPUT "ENTER YOUR AGE: ";A
```

If the user entered a nonnumeric character by mistake (say, the letter P), the program would continue to print ?REENTER until the user entered a number. We can eliminate this problem by using a string variable and VAL.

```
100 INPUT "ENTER AGE: ";A$
110 LET A = VAL (A$)
120 IF A = 0 THEN 100
```

If a letter is entered, its VALue would be 0 (zero). Line 120 successfully traps the error and returns the program to reenter the age.

STR$

STR$ does the opposite of VAL; that is, it changes a number into a string. The following is a typical example.

```
10 INPUT "ENTER ITEM CODE (4 CHAR MAX): ";IC
20 LET IC$ = STR$ (IC)
30 IF LEN (IC$) < > 4 THEN 10
40 IF LEFT$ (IC$,1) = "1" THEN GOTO 1000
   :
   :
```

Line 10 uses the numeric variable IC.

LIne 20 converts the numeric variable IC into a string variable IC$.

Line 30 checks the string's length.

Line 40 checks the leftmost character of the string for a specific number.

ASC

ASC is short for **ASCII,** which stands for American Standard Code for Information Interchange. Every key or combination of keys, such as CTRL with another key, produces a particular pattern or code. These patterns are represented by the ASCII decimal value. Appendix A lists the values and their equivalent characters. Refer to it as we discuss the following.

The ASCII value for the capital letter A is 65.

The ASCII value for the number 2 is 50.

The ASCII value for CTRL-T is 20.

The ASC function returns the ASCII value for the first character of (sexpr). The format is

ASC (sexpr)

where sexpr is equal to a string variable.

In a program, it could appear as

```
ASC (N$)
ASC ("JOHN")
```

Note that a string of characters such as JOHN must be enclosed in quotation marks. Therefore, quotation marks cannot appear within the string.

ASC can be valuable in trapping errors. We generally use it to compare strings in an IF THEN statement.

```
10 A$ = "APPLE": B$ = "APE"
20 IF A$ > B$ THEN PRINT "GREATER"
30 IF A$ < B$ THEN PRINT "LESSER"
RUN
```

Each string is compared to every other, one character at a time, left to right. When BASIC finds the first difference, the character that has a lower ASCII value is considered less than the other string. In the previous example, the condition in line 20 is met, so "GREATER" is displayed.

Which string is "LESSER"?

A$	B$
SALE	SALES

The answer is A$. The fifth character of string A$ is a null character (there is no character), and the fifth character of B$ is S. A null character is always less than a nonnull character.

Let's present a few practical error traps using the ASC function. Suppose you have an inventory program. One of your data fields is a product number. The product number must be a five-digit code beginning with a letter.

```
100 INPUT "ENTER PRODUCT NUMBER (5 CHAR): ";PN$
110 IF LEN (PN$) < > 5 THEN PRINT "ENTER A 5 DIGIT CODE": GOTO 100
120 IF ASC (PN$) < 65 OR ASC (PN$) > 90 THEN PRINT "FIRST
    CHARACTER MUST BE A LETTER.": GOTO 100
```

Line 110 checks the string length.

Line 120 ensures that a letter is the first character entered.

Another example:

```
100 INPUT "ENTER CHOICE A-D: ";A$
110 A = ASC(A$) - 64
120 IF A < 1 OR A > 4 THEN 100
130 ON A GOTO 200,300,400,500
140 :
150 REM REST OF PROGRAM
```

This example represents a menu-driven option selection. Line 100 requests that the user INPUT a letter. Line 110 makes A = ASC (A$) − 64. If the user chooses the letter A, then the ASCII equivalent of A is 65, so 65 − 64 = 1. The ASCII equivalent of B is 66, so 66 − 64 = 2, and so on.

Now that the response is converted to numbers, we can use a multibranching statement, as shown in line 130. Other examples of ASC are given under the MID$ function in this chapter.

CHR$

CHR$ stands for character string. The format is

CHR$ (aexpr)

where aexpr is equal to a number between 0 and 255.

This function returns the ASCII value that corresponds to the value of aexpr. For example,

```
10 PRINT CHR$ (65)
```

displays the capital letter "A." Appendix A lists the ASCII values.

CHR$ is the opposite of ASC, because it requires numbers rather than string variables inside parentheses. CHR$ used with a number displays the number's ASCII equivalent.

You can use CHR$ to send control characters to a printer. You can also use it to establish a CTRL-D character, as in

```
10 D$ = CHR$(4): REM CTRL-D
```

We can use it to ring the computer's bell, as in the following example.

```
20 INPUT "ENTER A NUMBER BETWEEN 1 AND 5: ";N$
30 IF N$ < "1" OR N$ > "5" THEN PRINT CHR$(7);"CAN'T YOU READ
   INSTRUCTIONS!": PRINT "TYPE A NUMBER BETWEEN 1 AND 5":
   GOTO 20
```

LEFT$

The LEFT$ function returns the leftmost character of a string. Its format is

LEFT$ (sexpr,aexpr)

where sexpr is equal to a string variable, and aexpr is equal to a numeric variable between 1 and 255.

In a program it would look like this:

```
20 N$ = "HILL"
30 PRINT LEFT$ (N$,2)
```

Line 20 sets the string variable N$ equal to the word HILL.

Line 30 PRINTs the two leftmost characters in the string.

The result is

```
HI
```

We can use LEFT$ to trap errors. It is especially effective with an INPUT statement that requires a yes or no response. For example,

```
10 INPUT "ENTER 1ST CHOICE: "C1$
20 INPUT "ENTER 2ND CHOICE: ";C2$
30 HOME: PRINT C1$:PRINT C2$
35 PRINT:PRINT
40 INPUT "IS THIS CORRECT (YES/NO): ";Y$
50 IF LEFT$(Y$,1) < > "Y" THEN 10
```

The user can answer Y, YES, YES SIR, or N, NO, NO WAY. The error trap in line 50 looks only at the leftmost character of the response. A LEFT$ function must include the $. If you omit it, APPLESOFT interprets LEFT as a numeric variable (LE) and displays a ?TYPE MISMATCH ERROR message.

RIGHT$

The RIGHT$ function is just like LEFT$, except that it returns the rightmost character of a string. The format is

RIGHT$ (sexpr,aexpr)

where sexpr is equal to a string variable, and aexpr is equal to a numeric variable between 1 and 255.

As with LEFT$, the "$" must be included.

RIGHT$ is not used often in programming, although it is sometimes handy for checking right-hand date or code number entries, etc. The following example shows one such use.

```
100 INPUT "ENTER TODAY'S DATE (EX: 071585): ";DT$
110 IF LEN (DT$) < > 6 THEN CALL -198: PRINT "ENTER A 6
    DIGIT DATE AS GIVEN IN EXAMPLE.": GOTO 100
120 D = VAL (RIGHT$(DT$,2)): IF D < > 85 THEN CALL -198:
    PRINT "PLEASE ENTER DATE FOR THIS YEAR": GOTO 100
```

Line 100 requests entry of a date.

Line 110 traps for correct length.

Line 120 checks for the correct year. The two rightmost characters, i.e., 85 are represented by RIGHT$(DT$,2). D is made equal to the VALue of the two string characters 85. As an arithmetic expression they can then be evaluated with a less than or greater than statement.

MID$

The MID$ function is one of your most versatile tools in programming. It can appear in the following forms:

```
MID$ (A$,2,4)
MID$ ("FRENCH TOAST",1,3)
MID$ (N$,X,1)
MID$ (N$,5)
```

Example

```
10 A$ = "APPLE PIE"
20 PRINT MID$ (A$,2,4)
```

Line 20 tells the computer to look at A$ and PRINT four characters beginning with the second character. This displays PPLE.

The first number (2) specifies which character to begin with, and the second number (4) specifies how many characters to print.

If you want to print everything from a particular character on, simply omit the number of characters to be printed to. For example,

```
PRINT MID$ (A$,3)
```

prints everything in A$, beginning with the third character.

MID$ is an efficient way to trap illegal characters. For example, suppose a program asks for a five-digit code for an inventory stock item. The codes can contain only numbers (0–9).

```
100 INPUT "ENTER ITEM CODE (5 CHAR): ";IC$
110 IF LEN (IC$) < > 5 THEN PRINT "ENTER 5 DIGITS": GOTO 100
120 FOR J = 1 TO LEN (IC$)
130 IF ASC (MID$(IC$,J,1)) < 48 or ASC (MID$(IC$,J,1))
    > 57 THEN PRINT CHR$(7); "ENTRY MUST BE NUMBERS ONLY.":
    GOTO 100
140 NEXT J
```

Line 120 creates a loop with LEN (IC$) as the upper limit.

Line 130 uses MID$ to check each character in the string individually. If any of the ASCII equivalents are invalid (e.g., the string includes something other than a number), the program returns to line 100.

Another example Suppose the item code could contain either letters or numbers, but not spaces. To trap for an illegal space, we could write the following:

```
100 INPUT "ENTER ITEM CODE (5 CHAR): ";IC$
110 IF LEN (IC$) < > 5 THEN PRINT "ENTER A 5 CHARACTER CODE":
    GOTO 100
120 FOR J = 1 TO LEN (IC$)
130 IF ASC (MID$ (IC$,J,1)) = 32 THEN 100
140 NEXT J
```

PROGRAMMING-RELATED ERRORS

One of the most common sources of error occurs in user INPUT statements. Poorly written or misleading prompts always guarantee problems. Prompts should clearly tell the user what the program wants.

Unfortunately, even when prompts seem clear to the programmer, the possibility of error still exists. Perhaps the user misinterprets the prompt or accidentally presses the wrong key.

Anytime I write a program, I ask my mother, a budding programmer, to test it. She seems to have an innate ability to always press some key other than the one requested and thereby discover unanticipated errors. I have concluded that if I can make my program "mother-proof," it is probably perfect. Having someone test your programs will help you see the programs more objectively.

Most user errors fall into three categories: beyond limits, incorrect type, and program interrupts. We will give a few examples of each of these. Then we will cover in more detail how to trap some of the more complex errors.

"Beyond Limit" Errors

Often, a program asks the user to enter a number. For example,

```
10  INPUT "ENTER A NUMBER BETWEEN 1 AND 10"; NU
```

A response of, say, 14 may lead to problems later. A properly written program always includes an error trap to keep the answer within limits.

```
10  INPUT "ENTER A NUMBER BETWEEN 1 AND 10"; NU
20  IF NU < 1 OR NU > 10 THEN CALL -198: PRINT: PRINT
    "NUMBER MUST BE BETWEEN 1 AND 10.": PRINT
    :GOTO 10
```

Line 20 traps the user's error by ensuring that the number entered is not less than 1 nor more than 10. If a number is beyond these limits, a bell rings, the user is instructed again what to enter. The program then returns to line 10 to let the user enter another number.

If the user enters a letter instead of a number, APPLESOFT automatically catches it and prints ?REENTER.

We can also place limits on user response in other ways:

```
10  PRINT "IS THIS CORRECT? (Y/N): ";: GET AN$: PRINT AN$
20  IF AN$ < > "N" AND AN$ < > "Y" THEN 10
```

or

```
10 PRINT "PRESS <E> TO EXIT, <P> TO PRINT OR": PRINT "<C> TO CATALOG
   DISK."
30 GET AN$: PRINT AN$
40 IF AN$ = "E" THEN END
50 IF AN$ = "P" THEN GOTO (line #)
60 IF AN$ = "C" THEN PRINT CHR$ (4); "CATALOG": GOTO (line #)
70 CALL -198: PRINT "PRESS ONE OF THE KEYS LISTED.":PRINT: GOTO 10
```

Both examples trap errors outside the specified limits.

"Incorrect Type" Errors

Sometimes a user may enter something other than what is requested, for example, a letter instead of a number or an unprotected comma or semicolon. Any of these would result in an ?EXTRA IGNORED message.

Suppose that in one of our previous examples we used a string variable NU$ instead of NU, as in

```
10 INPUT "ENTER A NUMBER BETWEEN 1 TO 10"; NU$
```

We could trap against a letter being entered by using **VAL,** that is,

```
10 INPUT "ENTER A NUMBER BETWEEN 1 TO 10"; NU$
20 IF VAL(NU$) = 0 THEN 10
```

If a letter were mistakenly entered, its VALue would be returned as zero. But what if zero is part of the number range, as in

```
10 INPUT "ENTER A NUMBER BETWEEN 0 AND 5: ";NU$
```

We could rewrite our error trap as

```
10 INPUT "ENTER A NUMBER BETWEEN 0 AND 5: ";NU$
20 IF VAL(NU$) = 0 AND NU$ < > "0" THEN 10
30 NU = VAL(NU$)
40 IF NU < 0 OR NU > 5 THEN 10
```

We can trap for the entries that follow a comma or semicolon. See Chapter 5 for ?EXTRA NOT IGNORED subroutines.

Interrupting a Program

The user can interrupt a program by pressing the RESET key or CONTROL-C. On the Apple II or II+ computers, a common mistake is to accidentally press RESET instead of RETURN, since the keys are close together.

You can set error traps in either case. Both of these interrupts produce error codes, which we will discuss in detail later in this chapter.

ERROR TRAP INPUT ROUTINES

Let's take a more detailed look at how to trap errors in an INPUT routine.

INPUT statements often allow many ways for the user to make a mistake. If, for example, you use the backspace key or left-arrow and go back too far, the cursor moves down to the next line from the INPUT prompt. Even though it still will take an INPUT, this isn't obvious to an inexperienced user.

Another confusing situation is when an INPUT routine accepts a string that is too long.

We can create an INPUT subroutine that processes each character as it is entered. This helps solve many problems, including limiting string length, limiting what characters can be entered, and handling the backspace key.

To begin, we will create a simple routine using GET. As each character is entered, it is added to the previously entered characters to form a string. The process is called *concatenation* (can anybody pronounce that?).

```
 10 PRINT "ENTER YOUR NAME: ";
 20 GOSUB 1010: NM$=A$:PRINT
 30 END

1000 REM NAME INPUT ROUTINE
1010 A$="": REM CLEAR A$
1020 GET CH$: REM INPUT FIRST CHARACTER
1040 A$ = A$ + CH$
1050 PRINT CH$;
1060 GOTO 1020
```

Line 1020 enters the character and adds it to string A$ in line 1040. Line 1050 displays the character. Notice the semicolon after CH$. This ensures that each character will be printed next to the previous one on the same line. Line 1060 sends the routine back to line 1020 for entry of the next character.

Of course, this program is not complete yet. All it does so far is increase the string A$ by one character each time it loops. We still need to consider what happens if the user presses a RETURN, arrow, backspace, ESCAPE, or CONTROL key.

We do that by adding

```
1030 IF ASC (CH$) < 32 THEN 1200
```

Now we are sending the routine outside the loop to handle those special keys.

```
1200 IF ASC(CH$) < > 8 THEN 1400: REM 8 IS BACKSPACE
1210 IF A$ = "" THEN 1020
1215 IF LEN (A$) = 1 THEN A$ = "": GOTO 1230
1220 A$ = LEFT$ (A$,LEN(A$) - 1)
1230 PRINT CH$;" ";CH$;
1240 GOTO 1020
```

Lines 1200–1240 handle the backspace key. If A$ is null (contains no characters), a backspace would move the cursor in front of the word, and then down to the next line. Line 1210 prevents this. If the string is null, then the program returns to line 1020.

If there is at least one character in A$, then line 1220 makes A$ equal to the string A$ without the last character. In other words, it eliminates the last character we typed.

When we press the backspace key in response to GET, the backspace key is stored as CH$. PRINT CH$ in line 1230 carries out the instruction to move the cursor back one space. This overwrites the character at that location with a space, and moves the cursor forward. To move the cursor back to its previous location, we must print CH$ again.

```
1230 PRINT CH$;" ";CH$;
```

In line 1240 the program again returns to line 1020 for the next character.

Note what could happen in line 1215. If LEN(A$) is 1, then LEN(A$) − 1 would be zero. Since LEFT$ doesn't work with zero, we bypass line 1220 by resetting A$ to "" and sending it back to GET.

We can handle the RETURN key with these lines:

```
1400 IF ASC(CH$) < > 13 THEN 1500
1410 RETURN
1500 GOTO 1020
```

Line 1400 instructs the computer to ignore any key or CONTROL character other than the RETURN key. If the RETURN is pressed, line 1410 instructs the computer to return to the GOSUB statement in line 20 and continue with the program. If any other key is pressed, line 1500 sends the program back to line 1020 to await another key entry.

Our complete routine looks like this:

```
  10 PRINT "ENTER YOUR NAME: ";
  20 GOSUB 1010: NM$=A$:PRINT
  30 END
1000 REM NAME INPUT ROUTINE
1010 A$="": REM CLEAR A$
```

```
1020 GET CH$: REM INPUT FIRST CHARACTER
1030 IF ASC(CH$) < 32 THEN 1200
1040 A$ = A$ + CH$
1050 PRINT CH$;
1060 GOTO 1020
1200 IF ASC(CH$) < < 8 THEN 1400: REM 8 IS BACKSPACE
1210 IF A$ = "" THEN 1020
1215 IF LEN (A$) = 1 THEN A$ = "": GOTO 1230
1220 A$ = LEFT$ (A$,LEN(A$) - 1)
1230 PRINT CH$;" ";CH$;
1240 GOTO 1020
1400 IF ASC(CH$) < > 13 THEN 1500
1410 RETURN
1500 GOTO 1020
```

Designing an INPUT statement that asks for more than one response is asking for trouble. For example,

```
20 INPUT "ENTER NAME AND AGE: ";N$,A
]RUN
ENTER NAME AND AGE: MARK
??
```

The user enters a name and presses the RETURN key. The double question mark indicates that another INPUT entry is desired, but an inexperienced user may not realize that.

Or perhaps the user enters the name, presses RETURN, and when the next question mark appears (not knowing what it's for), presses RETURN again.

```
]RUN
ENTER NAME AND AGE: MARK
??
?REENTER
```

In this case, the user isn't likely to know what ?REENTER means either. I could go on, but I think you can see the point. You can eliminate many programming problems by requesting only one variable assignment per statement, as in

```
20 INPUT "ENTER NAME: ";N$
30 INPUT "ENTER AGE: ";A$
```

Users often mistakenly enter letters instead of numbers in a numeric variable. To eliminate this error, assign all INPUT statements, whether string or numeric, to a string variable, e.g.,

```
30 INPUT "ENTER AGE: ";A$
40 LET A = VAL (A$)
50 REM REMAINING ERROR TRAPS
```

TRACE

As you gain experience in programming, your programs will become larger. Large programs often contain many subroutines. Naturally, this means the chances for errors are increased.

Below is a simple program using a subroutine. It demonstrates a common error, i.e., ?RETURN WITHOUT GOSUB ERROR IN (line #).

```
10 REM ERROR DEMONSTRATION
20 GOSUB 100: N$ = A$: PRINT N$
   :
   :
   :
100 INPUT "ENTER NAME: ";A$
110 IF A$ = "JOE" THEN H = 4
120 RETURN
```

Line 20 sends the program to the subroutine in lines 100–120. It then returns to line 20, continues, and executes the subroutine again. This time, when it reaches

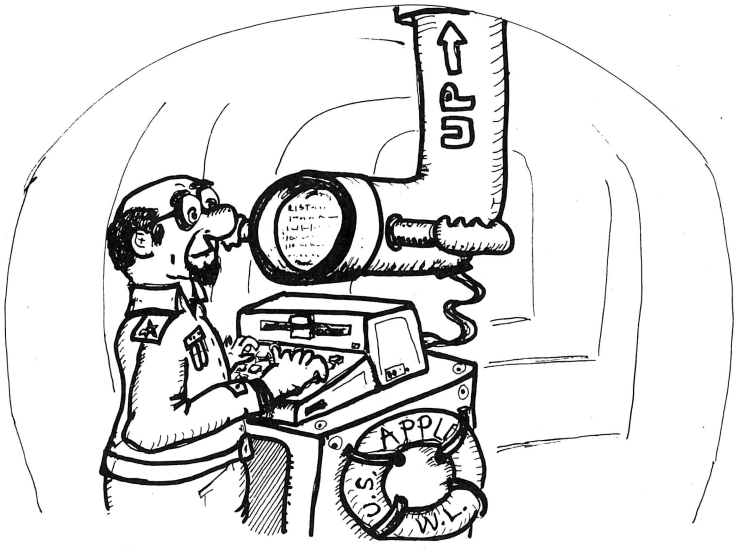


Figure 6-1 Subroutine.

RETURN, it displays the message "?RETURN WITHOUT GOSUB ERROR IN 120." GOSUB has not sent the program there. We can solve the problem by adding

```
30 END
```

Now when you RUN the program, it won't continue through the subroutine. When you have a very large program with numerous subroutines, detecting the source of a problem isn't so easy.

Fortunately, the Apple is equipped with a special feature called **TRACE**. You can use it in either immediate or deferred execution mode. That means you can either type the command directly into the computer from the keyboard before RUNning the program, e.g.,

```
]TRACE
RUN
```

or use the command within a program, e.g.,

```
10 TRACE
```

TRACE is a handy tool when you want to track down the source of a problem, as in the previous ERROR DEMONSTRATION example.

If we type TRACE from immediate mode before RUNning the program, it prints as follows:

```
]TRACE
]RUN
#10 #20 #100 ?ANSWER
#110 #120 #20 #20 ANSWER
#100 ?2ND ANSWER
#110 #120
?RETURN WITHOUT GOSUB ERROR IN 120
```

The #s are the line numbers of each statement as it is executed. It is now easy to follow the program's path as it is executed. We can backtrack from the error point to the problem source and correct it.

Once you have corrected the problem, you must stop the TRACE. Otherwise, it will TRACE the program every time you RUN it. To cancel TRACE type NOTRACE from either immediate or deferred execution mode.

SPEED

A program operates at a high speed. Sometimes we need to slow it down so we can see what is happening. This is especially important when we are using TRACE.

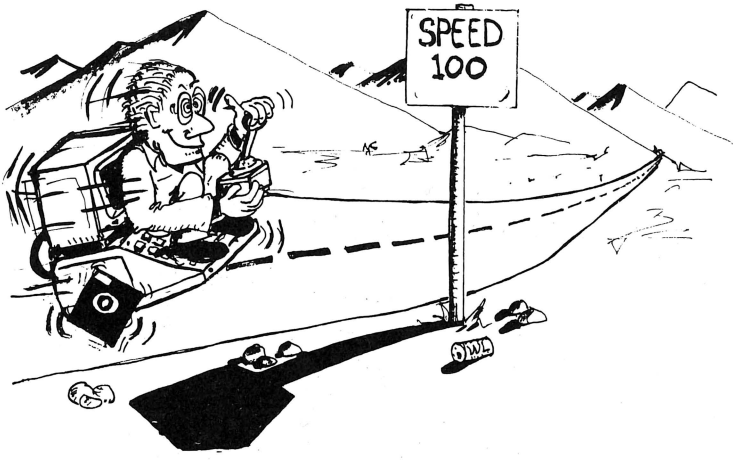


Figure 6-2

TRACE displays the line numbers as the program executes. However, they move by so quickly that you don't have a chance to read them. By imposing a **SPEED** limit on the program, we can slow it down.

The format is

SPEED = aexpr

where aexpr is equal to a number between 0 (slowest) and 255 (fastest).

The computer's normal operating SPEED is 255. The SPEED command may be used in immediate or deferred execution mode. Whatever SPEED is set stays in effect until you set a new one, even if you load a new program.

An example of SPEED is

```
50 SPEED = 100
60 PRINT "THIS PROGRAM HAS BEEN DESIGNED TO BE SLOW."
70 SPEED = 255
```

Note that line 70 resets the SPEED to 255.

To use SPEED in trapping errors, try

```
]TRACE
SPEED = 100
LIST 50,226
```

Experiment with different SPEEDs until you find one that you are most comfortable with. Don't forget to reset SPEED to its default value, 255.

DOS-RELATED ERRORS

MON and NOMON

A program with error traps can still have problems. The method that leads you to discover the problems or errors is called *debugging*.

The first computers were collections of hundreds of various tubes. Occasionally, an insect or bug got into the tubes, creating a shortcircuit in the system. The operator would have to find the bug and get rid of it. Thus, the origin of debugging.

One way to look for bugs is to monitor commands between the computer and the disk. Normally, these commands are not displayed, but we can see them with the DOS command **MON** (short for **MON**itor). **NOMON** (short for **NO MON**itor) cancels MON. Both work in immediate or deferred execution mode. As with all DOS commands in a program, you must precede MON and NOMON by a CTRL-D character. The following routine shows their typical use.

```
 90 D$ = CHR$(4): REM CTRL-D
100 PRINT D$;"MON C,I,O"
    :
    :
    :
150 PRINT D$; "NOMON C, I, O"
```

Figure 6-3 Debugging.

The three parameters are

C Commands to the disk (e.g. OPEN, READ)
I INPUT from the disk (when READing a file)
O Output to the disk (when WRITEing to a file)

Normally NOMON C,I,O is in effect.
The syntax for the commands is

MON [C] [,I] [,O]
NOMON [C] [,I] [,O]

You must use at least one of the three parameters with the commands, or else the command will be ignored. [C], [I], and [O] may be in any order, but they must be separated by commas.

There are seven ways to use MON:

Command	*What It Monitors*
MON C	commands to the disk
MON I	input from the disk
MON O	output to the disk
MON I,O	input from and output to the disk
MON C,I	commands to and input from the disk
MON C,O	commands to and output to the disk
MON C,I,O	commands to, input from, and output to the disk

MON stays in effect until you issue a NOMON, INT, or FP (firmware only) command or reboot the disk or press the RESET key.

MON can be helpful when you have problems READing or WRITEing from or to a file. It lets you see how the intermediate commands are being read.

USING ONERR GOTO ROUTINES

Sherlock Holmes never had it so good! The Apple has another special feature that not only aids in finding errors, but also lets you correct them without interrupting the program.

Each time an error occurs, the Apple stores a number (or *error code*) in memory address **222.** Each error generates a different number.

For example, if the error is a RETURN WITHOUT GOSUB, the Apple puts error code 22 into memory address 222. The code for a SYNTAX error is 16. A listing of both programming and DOS-related errors, their codes and most common causes are at the end of this chapter.

You can make this important information work for you by including an error code routine in your program, for example,

```
20 ONERR GOTO 8010
```

This line tells the computer that if any error occurs, regardless of the code, then go to line 8010 immediately. Note that the ONERR instruction should be one of the first lines in a program, to protect against errors as early as possible.

What's so special about the routine at line 8010? We can compare its function to that of the vice-principal in a junior high school. A teacher sends a student who causes a problem to the vice principal's office. After dealing with the problem, the vice-principal sends the student back to the classroom, and thereafter the student behaves (we hope) more acceptably.

When a program has a problem (error code) we can send it to an ONERR routine to be dealt with. The following example shows how this works.

```
8000 REM ONERR ROUTINES
8010 Y = PEEK (222)
8020 IF Y < > 5 THEN 8100
8030 PRINT D$; "CLOSE"
8040 CALL -198 : REM RING BELL
8050 HOME : PRINT "AN END OF DATA ERROR HAS OCCURRED."
8060 GOTO 8510

8100 IF Y < > 12 THEN GOTO 8510
8110 PRINT D$; "CLOSE"
8120 GOTO (some line # you want)

8500 REM PRINT ERROR LINE AND CODE
8510 PRINT "ERROR AT LINE: "; PEEK (218) + PEEK (219) * 256
8515 PRINT "ERROR CODE: ";Y
8520 END
```

Since we know that the error code is stored in memory location 222, we can PEEK there and find out what it is. Line 8010 sets the variable Y to the error code in memory location 222.

Line 8020 first checks to see if the error code is 5 (END OF DATA ERROR). If it is, the program continues to line 8030 and so on for corrective action. If the error code isn't 5, the program skips to line 8100. It then checks to see whether the error code is 12 (NO BUFFERS AVAILABLE). If it isn't 12, it skips to line 8510.

Since it isn't practical to write a solution for every error code specifically, we include lines 8510 and 8515 to catch all those not covered. These lines display each error's code and the program line number in which it occurs.

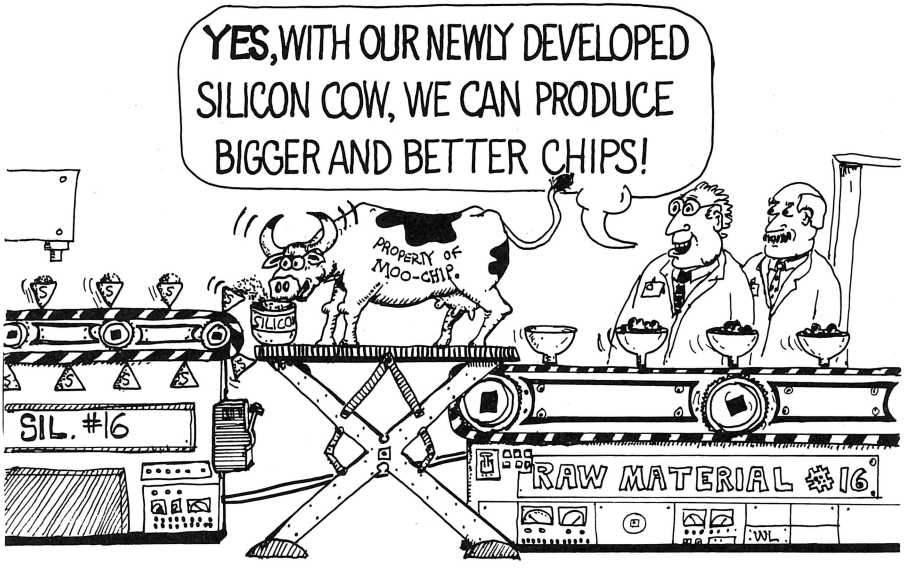

YES, WITH OUR NEWLY DEVELOPED SILICON COW, WE CAN PRODUCE BIGGER AND BETTER CHIPS!
PROPERTY OF MOO-CHIP
SIL. #16
RAW MATERIAL #16

Figure 6-4

. Remember that when you set an ONERR trap, an error will always cause the computer to proceed to the specified line number unless you tell it to do otherwise. If you want to disenable the error trap, so that normal error messages appear, you must clear the **ERRFLG** (error flag) by inserting a line like

```
370  POKE 216,0
```

You may compare this to deactivating a traffic light (e.g., from a school or industrial park) during hours when it is not needed. The line number you use may be different from the above example. Use this command when you no longer want the ERRFLG enforced. When you want an error trap again, simply type in on a program line ONERR GOTO (line #) once more.

One problem with Apple reference manuals is that, although they list error codes and causes of errors, there are no instructions on how to write error routines.

The following examples list common error codes and routines for using them.

DOS Error Code 4—Write Protected

```
8160  IF Y < > 4 THEN GOTO 8200
8170  PRINT "THIS DISK IS WRITE-PROTECTED. PLACE" : PRINT "ANOTHER IN THE
      DISK DRIVE.": PRINT "THEN PRESS ANY KEY.": GET AN$: PRINT
8180  GOTO ( line #)
```

or

```
8170 FLASH: PRINT "WARNING";:NORMAL:PRINT " :DISK HAS WRITE
     PROTECT":PRINT "TAB. REMOVE TAB OR GET ANOTHER DISK."
     :GET AN$: PRINT
```

or

```
8170 PRINT "WRITE PROTECT TAB ON DISKETTE": PRINT "IF YOU WANT
     TO SAVE/DELETE":PRINT "A FILE/PROGRAM, PLEASE REMOVE TAB.":
     GET AN$: PRINT
```

DOS Error Code 5—End of Data Error

```
8200 IF Y < > 5 THEN GOTO 8260
8210 PRINT "NO DATA FOR THAT FILE FOUND."
```

or

```
8210 PRINT "UNABLE TO LOCATE DATA BY THAT NAME."
```

or

```
8210 PRINT "END OF DATA. CHECK FILENAME": PRINT D$;
     "DELETE "; F$ (F$ = file name) : GOTO (line #)
```

DOS Error Code 6—File Not Found

```
8260 If Y < > 6 THEN 8300
8270 PRINT "NO FILE BY THAT NAME."
```

or

```
8270 PRINT "I CAN'T FIND THAT NAME ON THIS DISK. TRY":
     PRINT "A DIFFERENT NAME OR CHANGE DISKS."
```

or

```
8270 PRINT "FILE NOT FOUND!! SHOULD WE CREATE ONE (Y/N)? ";:
     GET AN$: PRINT AN$: IF AN$ = "Y" THEN GOTO (line #)
```

or

```
8270 PRINT "FILE "; FN$; " IS NOT IN THE CATALOG.":
     PRINT "DO YOU WANT TO INSERT A NEW DISKETTE?":
     PRINT "Y)ES, N)O : ";: GET AN$: PRINT AN$:
     IF AN$ = "N" THEN GOTO (line #)
8280 PRINT "PLEASE ENTER NEW DISK AND PRESS RETURN";:
     GET AN$:PRINT AN$: GOTO (line #)
```

DOS Error Code 8—I/O Error

```
8330 IF Y < > 8 THEN GOTO 8350
8335 PRINT "AN I/O ERROR HAS OCCURRED.":END
```

or

```
8335 FLASH: PRINT "WARNING!!";:NORMAL: PRINT "DISK DRIVE
     ERROR.":PRINT "VERIFY DISK IS INSERTED AND DOOR IS
     CLOSED."
```

DOS Error Code 9—Disk Full

```
8300 IF Y < > 9 THEN GOTO 8350
8310 PRINT "DISK FULL . . . SET UP A NEW DISK (Y/N): ";: GET
     AN$: PRINT AN$: GOTO (line #)
```

or

```
8310 PRINT "DISK FULL. PLACE ANOTHER DISK IN DRIVE."
```

or

```
8310 PRINT D$;"DELETE ";FN$
8320 FLASH: PRINT "WARNING!!";: NORMAL: PRINT " DISK IS
     FULL.": PRINT "YOUR OUTPUT FILE HAS BEEN DELETED.
     PLEASE INSERT": PRINT "ANOTHER DISK AND PRESS <RETURN>":
     GET AN$: PRINT AN$: GOTO (line #)
```

DOS Error Code 10—File Locked

```
8350 IF Y < > 10 THEN 8400
8360 PRINT "YOU ALREADY HAVE A FILE BY THAT NAME."
```

or

```
8360 PRINT "FILE ";F$;" IS LOCKED." : PRINT "OK TO
     OVERWRITE (Y/N): ";:GET AN$: PRINT AN$: IF
     AN$="Y" THEN PRINT D$;"UNLOCK";F$: PRINT D$;"SAVE ";F$:
     GOTO (line #)
8370 GOTO (line #)
```

DOS Error Code 11—Syntax Error

```
8000 If Y < > 11 THEN 8400
8010 CALL -198
8020 PRINT "NOT A LEGAL FILE NAME.": PRINT "TRY
     AGAIN. PRESS <RETURN>: ";: GET AN$: GOTO (line #)
```

DOS Error Code 12—No Buffers Available

```
8400 IF Y < > 12 THEN 8500
8410 PRINT D$;"CLOSE"
8420 GOTO (line #): REM SEND TO OPEN FILE COMMAND
```

For more information on why this error occurs, read the section on MAXFILES in Chapter 8.

Programming Error Code 254—Bad Response to an Input Statement

```
8380 IF Y = 254 THEN PRINT CHR$(7);"NUMERICAL ENTRY EXPECTED":
     PRINT: RESUME
```

or

```
8380 IF Y = 254 THEN PRINT "YOU HAVE MADE A DATA ENTRY ERROR.":PRINT
     "PLEASE TRY AGAIN.": RESUME
```

or

```
8380 IF Y = 254 THEN PRINT CHR$(7);:RESUME
```

Programming Error Code 255—CTRL C Interrupt Attempted

```
8370 IF Y = 255 THEN FLASH:PRINT "WARNING: ";:NORMAL: PRINT
     "DO NOT PRESS CTRL-C OR":PRINT " RESET. DO YOU WANT TO
     ABORT THE":INPUT "PROGRAM (Y/N)? ";AN$
```

or

```
8370 IF Y = 255 THEN PRINT CHR$(7);: RESUME
```

DOS Error Code 56—Reset

If your Apple has auto-start ROM, then you can use this routine to produce an error message any time the RESET key is pressed.

```
  10 POKE 1010, 233: POKE 1011, 242: CALL -1169
  20
  40
  50      (rest of program)
  60
8200 IF Y = 56 THEN CALL 43089: RESUME
```

This protects against accidental resets. If you want to give the user an option to abort the program, use program line 8370 suggested under PROGRAMMING ERROR CODE 255, with the following exception:

```
8200  IF Y = 56 THEN . . .
```

You could replace RESUME in line 8200 with NEW, or GOTO (line #) or RUN, etc.

Some DOS errors can be avoided if you keep in mind their limitations, as follows:

	DOS Limitations	
	Mimimum Value	**Maximum Value**
VOLUME NUMBER	0	254
DRIVE NUMBER	1	2
SLOT NUMBER	1	7
RANGE ADDRESS	0	65,535
RANGE LENGTH	1	65,535
RECORD LENGTH	1	32,767
RECORD NUMBER	0	32,767
BYTE NUMBER	0	32,767

HINTS

Hint 1

The same type of error can occur in different parts of a program. Thus if you set up an error routine to return to a particular line number, the program may not handle all occurrences of the error.

There is a way to take care of this situation. As an example, we will use error code 5.

```
8000  ER = PEEK (222) : LN = PEEK (218) + PEEK (219) * 256

8010  IF ER = 5 AND LN = 580 THEN 1040
8020  IF ER = 5 AND LN = 780 THEN 2275
```

We have set the error code to the variable ER and the line number to LN in line 8000. By using IF THEN statements, we gain further control over where the program returns after an error correction.

Hint 2

Sometimes when working with text files, you may attempt to OPEN a file that doesn't exist. Perhaps you misspelled the filename. DOS will create the file

disk if it doesn't exist. But it quickly returns an END OF DATA message, because it attempts to READ a file that has nothing in it.

A catalog of the disk would show that file occupying one sector; that is,

```
T 001 FILENAME.
```

To prevent accidental file creation, use the VERIFY command before opening any text files.

```
690 ONERR GOTO 8000
700 PRINT D$;"VERIFY";F$ : REM F$ = FILE NAME
710 PRINT D$;"OPEN";F$
720 rest of program
```

If the file doesn't exist, the computer will generate error code 6, "FILE NOT FOUND." You can then either create the file or correct the misspelling.

DOS ERROR CODES

Code	*DOS Message*	*Most Common Cause*
1	LANGUAGE NOT AVAILABLE	Applesoft not on diskette
2, 3	RANGE ERROR	Command parameter too large
4	WRITE PROTECTED	Write-protect tab on diskette
5	END OF DATA ERROR	READing beyond end of text file
6	FILE NOT FOUND	File misspelled, or not on diskette
7	VOLUME MISMATCH	Wrong volume parameter
8	I/O ERROR	Door open, or diskette not INITialized
9	DISK FULL	Too many files on diskette
10	FILE LOCKED	Attempt to overwrite a LOCKed file
11	SYNTAX ERROR	Bad file name, parameter, or comma
12	NO BUFFERS AVAILABLE	Too many text files OPEN
13	FILE TYPE MISMATCH	Diskette file doesn't match command
14	PROGRAM TOO LARGE	Insufficient Apple memory available
15	NOT DIRECT COMMAND	Command must be in a program

PROGRAM ERROR CODES

Code	*DOS Message*	*Most Common Cause*
0	NEXT without FOR	Variable in NEXT statement does not match variable in FOR statement or NEXT issued when no FOR in effect.
16	Syntax error	Missing parenthesis, illegal character, incorrect punctuation in expression or program line.
22	RETURN without GOSUB	RETURN encountered without a GOSUB having been executed.
42	Out of DATA	READ statement tried to read more from the DATA statements than was there.
53	Illegal Quantity	A parameter passed to a math or string function that is out of range.
69	Overflow	Calculation's results too large to be represented in BASIC's number format.

PROGRAM ERROR CODES (*continued*)

Code	*DOS Message*	*Most Common Cause*
77	Out of Memory	Program too large; too many variables; FOR loops nested more than 10 levels deep; GOSUB's nested more than 24 levels deep; too complicated an expression; parentheses nested more than 36 levels deep; attempt to set LOMEM: too high; attempt to set LOMEM: lower than present value; attempt to set HIMEM: too low.
90	Undefined Statement	Attempt to GOTO, GOSUB, or THEN to a statement line number which does not exist.
107	Bad Subscript	Attempt to reference an array element outside the dimensions of the array.
120	Redimensioned Array	Attempt to redimension an array which was has already been dimensioned somewhere else in the program.
133	Division by Zero	Dividing by zero is an error.
163	Type Mismatch	Left-hand side of an assignment statement was a numeric variable and the right-hand side was a string, or vice versa; or a function which expected a string argument was given a numeric one or vice versa. Example: Wrong: A$ = LEN (S$) Correct: A = LEN (S$)
176	String Too Long	Trying to create a string more than 255 characters long by concatenation.
191	Formula Too Complex	More than two statements of the form IF "XX" THEN were executed.
224	Undefined Function	Reference to user-defined function that had never been defined.
254	Bad Response to an INPUT Statement	A numerical entry was expected.
255	Ctrl C Interrupt Attempted	Ctrl C was used to break out of program.

OTHER ERROR MESSAGES

Can't Continue

An attempt to continue a program when non existed, or after an error occurred, or after a line was deleted from or added to a program.

Illegal Direct

You cannot use an INPUT, DEF FN, GET, or DATA statement as an immediate-execution command.

7 What's the Routine?

INTRODUCTION

This chapter is a collection of helpful routines. Some routines may refer you to a chapter for additional understanding. This will be indicated in parentheses at the end of each routine or description, e.g. (Chapter 2).

CATALOG LISTER

Type these POKEs into your "Hello" program. After booting the disk you can then CATALOG it by typing only the ampersand "&" (shift 6) and pressing RETURN.

```
10  POKE 1014,110: POKE 1015,165
```

(Chapter 2 and Chapter 6)

AMPERSAND KILL—CATALOG UTILITY

Have you ever CATALOGed a disk only to see the file name you want scroll off the top of the screen before you can look at it? This program utilizes the CATALOG LISTER, but introduces an additional feature, CTRL-C CATALOG KILL. Type "&" to CATALOG the disk and display the first page of files. Then press CTRL-C to "kill" the CATALOG. This freezes the list. You can then find the filename you want and LOAD it.

```
10 POKE 1013,76: POKE 1014,110: POKE 1015,165
20 KEY$ = CHR$ (3)
30 T = 256:DOS = PEEK (978) * T + 4410:AE = 174 - (157 -
   PEEK (978)):L = DOS + 3119: POKE DOS,L - INT (L / T) * T:
   POKE DOS + 1, INT (L / T)
40 POKE L + 0,32: POKE L + 1,12: POKE L + 2,253: POKE L + 3,201:
   POKE L + 4, ASC (KEY$) + 128: POKE L + 5,208: POKE L + 6,3:
   POKE L + 7,76: POKE L + 8,44: POKE L + 9,AE: POKE L + 10,76:
   POKE L + 11,60: POKE L + 12,AE: POKE L + 13,0
50 HOME : VTAB 10: HTAB 5: PRINT "TYPE AN '&' AND THEN A (CR)
   RETURN", "FOR A CATALOG. A 'CTRL. C' WILL MAKE", "A CLEAN
   BREAK IN THE CATALOG. ANY OTHER", "KEY WILL CONTINUE THE
   CATALOG."
```

CURSOR DASH

This program changes the old familiar flashing cursor into a dashing fellow.

```
10 REM CURSOR DASH
20 :
30 :
40 POKE 768,72: POKE 769,169: POKE 770,223: POKE 771,145: POKE
   772,40: POKE 773,104: POKE 774,76: POKE 775,27: POKE 776,253:
   POKE 56,0: POKE 57,3: CALL 1002: END
70 REM THIS PROGRAM CHANGES THE
80 REM CURSOR. POKE DIFF NO'S
90 REM IN POKE 770.
```

POKEing different ASCII values in POKE 770 creates new types of cursors.

APPLEBETIZE

There are several ways to alphabetize words or sort numbers. Words (actually strings that represent words) are usually assigned to array string variables. These arrays are then sorted and displayed.

The following examples show several types of sort routines. If you plan to use a sort routine in a program, change the variables to match whatever the program uses. For example, in the program below, the variables NW and W$(T) might, in your program, be R1 and NM$(T).

Alphabetize

```
 10 REM ALPHABETIZE
 20 REM PUBLIC DOMAIN - BY RICHARD VERLAQUE

100 HOME: PRINT "OUTPUT TO PRINTER? ";: GET A$ : PRINT A$
110 IF A $ = "Y" THEN PR = 1
120 INPUT "NUMBER OF WORDS: ";NW
130 IF NW < 2 OR NW < > INT (NW) THEN 120
140 PRINT: DIM W$(NW)
150 FOR T = 1 TO NW
160 INPUT "WORD ";W$(T)
170 NEXT T
180 C = O: FOR X = 1 TO NW - 1
190 IF PR THEN PRINT CHR$(4);"PR#1" : REM TURN PRINTER ON
200 IF W$(X) < = W$(X+1) THEN 250
210 T$ = W$(X)
220 W$(X) = W$(X+ 1)
230 W$(X + 1) = T$
240 C = C + 1
250 NEXT X
260 IF C > 0 THEN 180
270 FOR X = 1 TO NW
280 PRINT "WORD NBR ";X;" ";W$(X)
290 NEXT X: PRINT
300 IF PR THEN PRINT CHR$(4);"PR#0" : REM TURN PRINTER OFF
310 PRINT "ENTER MORE WORDS? ";: GET A$ : PRINT A$
320 IF A$ = "Y" THEN CLEAR: GOTO 100
330 END
```

Bubble Sort

```
 10 HOME: INPUT "ENTER NUMBER OF WORDS: ";N
 20 FOR J = 1 TO N
 30 PRINT "ENTER WORD ";J;": ";
 40 INPUT A$(J)
 50 NEXT J

 60 REM SORT
 70 FOR X = 1 TO N - 1
 80 FOR J = 1 TO N - 1: IF A$(J) < = A$(J + 1) THEN 100
 90 K$ = A$(J): A$(J) = A$(J + 1): A$(J + 1) = K$
100 NEXT J : NEXT X
```

```
200  REM PRINT SORTED LIST

210  FOR X = 1 TO N
220  PRINT "#";X;"  ";A$(X)
230  NEXT X
240  END
```

Shell-Metzner Sort

```
 10  HOME: INPUT "HOW MANY NUMBERS TO SORT: ";N
 20  FOR J = 1 TO N
 30  PRINT "ENTER NUMBER ";J;": ";
 40  INPUT A(J)
 50  NEXT J

 60  REM SORT

 70  K1 = N + 1
 80  K1 = INT (K1 /2): IF K1 = 0 THEN 200
 90  K2 = N - K1: J = 1
100  X = J
110  K3 = X + K1: IF A(X) < A(K3) THEN 150
120  K4 = A(X): A(X) = A(K3): A(K3) = K4:: X = X - K1:
     IF X > 1 THEN 110
150  J = J + 1: IF J > K2 THEN 80
160  GOTO 100

200  REM PRINT SORTED LIST

210  FOR X = 1 TO N
220  PRINT "#";X;"  ";A(X)
230  NEXT X
240  END
```

COLUMN ALIGNMENT

Ever need to align numbers in a column? The following programs show you how.

Simple Alignment

This program will right-justify any number entered.

```
10  REM COLUMN ALIGNMENT RIGHT JUSTIFY
20  INPUT "ENTER NUMBER: ";N
25  HTAB 30
```

```
30 PRINT SPC(N<1000);SPC(N<100);SPC(N<10);N
40 GOTO 20
```

Changing the HTAB in line 25 moves the numbers to that column.

The following two programs format entries to dollars and cents.

Dollars and Cents 1

```
 10 HOME
 20 :
 30 PRINT "ENTER AMOUNT: ";
 40 INPUT Z
 50 :
 60 GOSUB 1010
 70 :
 80 REM PRINTS NUMBER TO SCREEN
 90 :
100 POKE 36,35 - LEN (X$): PRINT X$
110 GOTO 30
    :
    :
1000 REM SUB CALCULATES ALIGNMENT
1010 IF Z < .10 THEN LET X$ = STR$ (Z): RETURN
1020 Z = INT (Z * 100 + .05) / 100: Z = Z * 100: Z$ = STR$(Z):
     ZZ$ = RIGHT$ (Z$,2): IF ABS (Z) > = 100 THEN ZQ$ =
     LEFT$ (Z$, LEN (Z$) - 2): X$ = ZQ$ + "." + ZZ$ : RETURN
1030 ZZ$ = RIGHT$ (Z$,2) : ZQ$ = "0": X$ = ZQ$ + "." + ZZ$: RETURN
```

Line 100 determines the column location of the numbers. POKE 36 controls the horizontal cursor position. If you want the numbers to appear in a different column, change the second number (35) in the POKE statement to a number BETWEEN 0 and 39.

Dollar and Cents 2

```
 10 REM COLUMN ALIGNMENT
 20 REM NEGATIVE #'S INCLUDED
 30 :
 40 :
 50 REM H = HTAB OF RIGHTMOST COLUMN LOCATION OF AM
 60 :
 70 H = 35
 80 :
 90 :
100 HOME
110 INPUT "ENTER AMOUNT: ";AM
```

```
120 GOSUB 1020
130 GOTO 110
140 :
    :
    :
1000 REM ALIGNMENT SUB
1010 :
1020 IF AM = 0 THEN AM$ = "0.00": GOTO 1070
1030 :
1040 SG = 1: SG$ = " ": IF AM < 0 THEN SG = -1: SG$ = "-"
1050 AM$ = LEFT$ ( STR$ (AM * SG + .005), LEN ( STR$ (AM *
     SG + .005)) - 1) + SG$
1060 :
1070 HTAB H - LEN (AM$): PRINT AM$
1080 RETURN
```

This column alignment routine is more complete. It handles positive and negative numbers, as well as a zero (0) entry. The variable H determines the HTAB of the rightmost column in which you want the number to appear. You can reset the value of H (line 70) to any number between 1 and 40.

AN AVERAGING ROUTINE

The following routine averages a group of numbers.

```
100 REM AVERAGING
110 HOME
120 PRINT "ENTER NUMBERS AS REQUESTED."
125 PRINT "ENTER 'E' TO EXIT."
130 PRINT:PRINT
140 FOR X = 1 TO 100: REM UP TO 100 NUMBERS CAN BE AVERAGED
150 PRINT "ENTER NUMBER ";X;":";: INPUT N$
160 IF N$ = "E" THEN 200
170 IF VAL (N$ ) < = 0 THEN VTAB PEEK (37): CALL -958:
    CALL -198: GOTO 150
180 T = T + VAL (N$)
190 NEXT X
200 X = X - 1
210 PRINT : PRINT "AVERAGE OF ";X;" NUMBERS: ";INT (T/X)
220 END
```

If you are working with dollars and cents, change line 210 to INT (100 * T/X)) / 100.

MORTGAGE CALCULATION

You can use this program to determine monthly principal and interest payments on a mortgage.

```
 10 REM MORTGAGE CALCULATION
 20 HOME
 30 INPUT "ENTER THE LOAN PRINCIPAL: ";P
 40 INPUT "ENTER INTEREST RATE (EXP. 10, 12.5): ";I
 50 INPUT "ENTER LIFE OF THE LOAN IN YEARS: ";Y
 60 INPUT "HOW MANY PMTS DO YOU MAKE EACH YEAR: ";N
 70 R = ((I / 100) * P/N) / (1 - 1 / (( I / 100 )
    / N + 1 ) ^ (N * Y ))
 80 PRINT: PRINT
 90 PRINT "REGULAR PAYMENTS OF $";INT (R * 100 + .5)/100;"."
100 PRINT "MADE ";N;" TIMES EACH YEAR"
110 PRINT "ARE REQUIRED TO SATISFY"
120 PRINT "A LOAN OF $";P;" FOR A PERIOD OF ";Y;" YEARS"
130 PRINT "AT AN INTEREST RATE OF ";I;" PERCENT PER ANNUM."
140 END
```

PROBABILITY

"It will probably happen!" What does that mean? You hear it all the time. Here's a program that, given the number of possibilities and the number of trials, will produce the probability of having at least one duplicate.

```
 10 REM PROBABILITY
 20 HOME: PRINT TAB ( 12) "PROBABILITY": PRINT: PRINT
 30 PRINT "ENTER TOTAL NUMBER OF POSSIBILITIES (EXP. SAME"
 40 PRINT "CARDS-52, DAYS OF WEEK-7, SAME BIRTHDAYS-365)"
 50 INPUT ": ";B$
 60 IF VAL (B$) > 1 THEN 100
 70 PRINT "PLEASE FOLLOW INSTRUCTIONS!"
 80 FOR X = 1 TO 1000: NEXT X
 90 GOTO 20
100 PRINT "ENTER THE NUMBER OF TRIALS."
110 INPUT ": ";C$
120 A =1 : E = VAL (B$): F = VAL (B$)
130 FOR B = 1 TO VAL (C$)
140 A = A * (E/F): E = E - 1
150 NEXT B
160 Q = 1 - A
170 HOME
```

```
180 PRINT "IF THE TOTAL NUMBER OF POSSIBILITIES"
190 PRINT "IS ";B$;" AND THE NUMBER OF TRIALS IS ";C$
195 PRINT : PRINT
200 PRINT "THE PROBABILITY OF AT LEAST ONE BEING A"
202 Q = INT (.5 + 100 * Q)
210 PRINT "DUPLICATE IS ";Q;" PERCENT."
220 END
```

Source: Public Domain Library, submitted by R.A. Justice.

ASCII TABLE

Here's how to generate your own ASCII table.

```
10 FOR X = 0 TO 31
20 FOR A = 0 TO 96 STEP 32
30 N = X + A
40 HTAB 1 + A / 4 + 5 * (A > 0)
50 PRINT SPC ( (N < 100) + (N < 10))N"";
60 IF N < 32 THEN PRINT "CTRL-"; CHR$ (N + 64);: GOTO 80
70 PRINT CHR$ (N);
80 IF A = 96 THEN PRINT ""
90 NEXT A : NEXT X
```

PADDLES OR JOYSTICK CONNECTED?

If you want to determine whether the paddles or joystick are connected in the gameport connector, try this:

```
10 IF PEEK (49249) AND PEEK (49250) > 127 THEN
   PRINT "NO PADDLES IN"
20 PRINT "PADDLES ARE CONNECTED."
```

(Chapter 5)

PADDLE TESTER

Now that you have checked the paddle or joystick connection, RUN this program. It checks whether the range and paddle buttons are working correctly.

```
10 KBD = - 16384:STB = - 16368:B1 = - 16287:B2 = - 16286
20 TEXT : HOME : HTAB 14: PRINT "[PADDLE TEST]"
```

```
 30 VTAB 3: HTAB 5: PRINT "PDL (0)   PB(0)    PDL(1)   PB(1)"
 40 VTAB 10: HTAB 8: PRINT "PRESS ANY KEY TO STOP"
 50 GOSUB 100: REM READ PADDLES
 60 GOSUB 190: REM DELAY HARDWARE
 70 G = PEEK (KBD): IF G < 128 THEN 50
 80 POKE STB,0: VTAB 10: HTAB 8: PRINT CHR$ (7)"PROGRAM STOPPED BY
    USER": HTAB 8: PRINT "CONTINUE?";: GET G$: IF G$ < > "N" THEN
    VTAB 10: HTAB 1: CALL - 958: GOTO 40
 90 END
100 P$ = STR$ ( PDL (0))
110 GOSUB 200
120 B1$ = "OFF": IF PEEK ( - 16287) > 127 THEN B1$ = " ON"
130 VTAB 5: PRINT "      "P$;: HTAB 14: PRINT B1$;
140 P$ = STR$ ( PDL (1))
150 GOSUB 220
160 B1$ = "OFF": IF PEEK ( - 16286) > 127 THEN B1$ = " ON"
170 PRINT "      "P$;: HTAB 30: PRINT B1$
180 RETURN
190 FOR I = 0 TO 3
200 NEXT I
210 RETURN
220 IF LEN (P$) < 3 THEN P$ = "0" + P$: GOTO 220
230 RETURN
```

Source: Public Domain Library by Peter Wyspianski.

UNDELETE LOST FILES

Have you ever accidentally DELETEd a file and wished you could undo the terrible deed? These two programs can recover the file on two conditions: It must be the last file you DELETEd, and you can't have SAVEd a file on the disk since the deletion. Otherwise, your file is gone forever!

"Undelete" Program 1

```
10 REM UNDELETE INSTRUCTIONS
20 R$ = CHR$(4) + "RUNUNDELETE"
30 HOME : COLOR= 10: VLIN 0,47 AT 0: VLIN 0,47 AT 39:
   HLIN 0,39 AT 0: HLIN 0,39 AT 46
40 VTAB 24: HTAB 8: PRINT " HIT ANY KEY TO CONTINUE ";: VTAB 3:
   HTAB 2
50 POKE 32,1: POKE 33,38: POKE 34,1: POKE 35,23
60 PRINT
```

```
 70 VTAB 12: PRINT "          <<< U N D E L E T E >>>"
 80 W = 100: GOSUB 320: POKE - 16368,0
 90 CALL - 912: VTAB 13: PRINT "              BY L. GUST":
    W = 50: GOSUB 320
100 POKE - 16368,0
110 HOME : PRINT "       I N S T R U C T I O N S": PRINT :
    PRINT "THIS PROGRAM CAN RETRIEVE A FILE OR": PRINT "PROGRAM
    THAT HAS BEEN DELETED FROM AN": PRINT "INITIALIZED DOS 3.3
    DISK."
120 W = 200: GOSUB 320
130 PRINT : PRINT "THE DISK SHOULD BE UNALTERED--NO";:W = 100:
    GOSUB 320; PRINT "SAVES SHOULD BE PERFORMED ON THE": PRINT "DISK.":
    GOSUB 320
140 PRINT : PRINT "IF YOU HAVE SAVED A FILE SINCE YOU": PRINT
    "HAVE DELETED ONE, THE CHANCES THAT": PRINT "THE ORIGINAL
    CAN BE RECOVERED ARE": PRINT "VERY, VERY SLIM.";:W = 125:
    GOSUB 320
150 PRINT : GOSUB 320: PRINT "(IT IS POSSIBLE, SO TRY THE
    PROGRAM": PRINT "--IT JUST MIGHT WORK.)"
160 W = 400: GOSUB 320: HOME
170 PRINT : PRINT
180 PRINT "THE PROGRAM WILL PROMPT YOU FOR A": PRINT "FILE
    NAME. THIS NAME MUST BE LESS": PRINT "THAN 28 CHARACTERS
    LONG. IF YOUR": PRINT "FILE HAS A LONGER NAME, JUST TYPE"
190 PRINT "27 CHARACTERS OF THE NAME."
200 W = 200: GOSUB 320
210 PRINT : PRINT "THE DISK WILL START TO SPIN IN A": PRINT
    "SECOND OR TWO. IF THE PROGRAM CANNOT": PRINT "FIND YOUR
    FILE, IT WILL TELL YOU.": GOSUB 320
220 PRINT : PRINT "IF IT FINDS YOUR PROGRAM, IT WILL": PRINT
    "REVIVE IT AND ALSO LOCK IT TO PREVENT": PRINT "ANY FURTHER
    MISHAPS.": GOSUB 320
230 PRINT : PRINT "TO EXIT NOW TYPE THE ";: INVERSE : PRINT
    "ESCAPE";: NORMAL : PRINT " KEY."
240 PRINT : PRINT "TO RUN <UNDELETE> HIT THE ";: INVERSE : PRINT
    "RETURN";: NORMAL : PRINT " KEY."
250 POKE - 16368,0
260 VTAB 22: HTAB 20: GET A$
270 IF A$ = CHR$ (27) THEN TEXT : HOME : END
280 IF A$ = CHR$ (13) THEN TEXT : HOME : PRINT : PRINT R$: END
290 FOR A = 1 TO 20:Z = PEEK ( - 16336): NEXT : GOTO 260
300 POKE - 16368,0
310 END
320 A = 0: POKE - 16368,0
330 A = A + 1: IF A > W THEN RETURN
340 IF PEEK ( - 16384) > 127 THEN RETURN
350 GOTO 330
```

"Undelete" Program 2

```
10 REM UNDELETE PROGRAM
20 HOME : VTAB 10: PRINT "WHAT IS THE FILE YOU WANT TO RECOVER?":
   VTAB 12: PRINT "> . . . . . ";: HTAB 1: INPUT
   ">";N$: IF LEN (N$) > 28 THEN 20
30 LF = 0
40 IF LEN (N$) = 0 THEN VTAB 12: PRINT ">>>          T E R M I N A
   T E D      <<<": END
50 Q$ = "": FOR A = 1 TO LEN (N$):Q$ = Q$ + CHR$ ( ASC ( MID$
   (N$,A,1)) + 128): NEXT
60 N$ = Q$
70 GOSUB 440
80 TRACK = 17: SECTR = 15
90 POKE START + 12,TRACK
100 POKE START + 13,SECTR
110 POKE START + 20,R
120 CALL 8192
130 LTRACK = PEEK (BUF + 1)
140 LSECTR = PEEK (BUF + 2)
150 IF LTRACK + LSECTR = 0 THEN LF = 1
160 A = 11
170 N1$ = ""
180 IF PEEK (BUF + A) = 255 THEN 240
190 A = A + 35
200 IF A < 223 THEN 180
210 IF LF = 1 THEN PRINT : PRINT "FILE IS EITHER NOT DELETED OR
   ELSE": PRINT "IT DOESN'T EXIST. PERHAPS YOU MISTYPED IT.":
   PRINT "PERHAPS YOU HAVE SAVED A FILE ON THIS": PRINT "DISK SINCE
   THE DELETION.": PRINT: PRINT: END
220 SECTR = SECTR - 1: IF SECTR = 0 THEN LF = 1: GOTO 210
230 GOTO 90
240 FOR B = 3 TO 31
250 N1$ = N1$ + CHR$ ( PEEK (BUF + A + B))
260 NEXT B
270 IF LEN (N1$) < 2 THEN 410
280 IF ASC ( RIGHT$ (N1$,1)) = 160 THEN N1$ = LEFT$ (N1$, LEN
   (N1$) - 1): GOTO 270
290 IF N1$ < > N$ THEN 190
300 PRINT " ************************************ ": PRINT
          "          F O U N D  I T          ": PRINT
          " ************************************ "
310 TN = PEEK (BUF + A + 32): POKE BUF + A,TN: POKE BUF +
   A + 32,160
320 L = PEEK (BUF + A + 2)
330 IF L < 128 THEN L = L + 128
340 POKE BUF + A + 2,L
```

```
350 POKE START + 20,W
360 POKE START + 12,TRACK
370 POKE START + 13,SECTR
380 CALL 8192
390 PRINT : PRINT :
    PRINT " ****************************************": PRINT
          "                RECOVERED               ": PRINT
          " ****************************************": PRINT
    : PRINT
400 PRINT CHR$(4);"CATALOG": END
410 IF N$ = N1$ THEN 300
420 GOTO 190
430 END
440 START = 8192:BUF = 9000: FOR A = START TO START + 28
450 READ V: POKE A,V: NEXT
460 DATA 169,32,160,8,32,217,3,96,1,96,1,0,17,15,25,32,
    40,35,0,0,1,0,0,96,1,0,1,239,216
470 R = 1:W = 2
480 RETURN
```

Source: Public Domain Library.

DELETE ARRAYS

Add these lines to your program to delete all arrays. It will let you reDIMension any or all arrays without error.

```
300 POKE 109, PEEK (107): POKE 110, PEEK (108)
310 X = FRE(0): REM STRING "HOUSE-CLEANING"
```

PEEK 0

One way to avoid seeing the same instructions or introductions each time you RUN a program is to add POKE 0,1 after they appear the first time, as in

```
700 REM INSTRUCTIONS
    :
    :
    :
800 POKE 0,1
810 RETURN
```

Then at the beginning of the program add

```
20 IF PEEK(0) = 1 THEN 40
30 GOSUB 700: REM PRINT INSTRUCTIONS
40 REM REST OF PROGRAM
   :
   :
   :
```

If the condition of line 20 is met, then the computer skips the instruction subroutine in line 30.

TIME CLOCK

Need a time clock? This program counts off "second" to none.

```
10 HOME: PRINT TAB (15) "TIME CLOCK"
20 VTAB 22
30 PRINT "TIME: ";T
40 FOR TIME = 1 TO 1000: NEXT
50 T = T + 1
60 VTAB 22
70 S = PEEK (-16336) + PEEK (-16336)
80 PRINT "TIME: ";T
90 GOTO 40
```

COUNTDOWN CLOCK

Unlike the time clock, which counts up, the countdown clock counts down. An alarm in lines 250–270 indicates when the time is up.

```
 10 REM COUNTDOWN CLOCK
 20 :
 30 :
 40 HOME: PRINT TAB ( 11)"COUNTDOWN CLOCK"
 50 :
 60 PRINT: PRINT
 70 INPUT "ENTER NUMBER OF SECONDS: ";SE
 80 :
 90 VTAB 22
100 PRINT "TIME: ";SE
110 :
```

```
120 FOR TIME = 1 TO 1000: NEXT TIME
130 :
140 SE = SE - 1
150 :
160 REM CLICK
170 :
180 S = PEEK (-16336) + PEEK (-16336)
190 :
200 VTAB 22
210 CALL -868: PRINT "TIME: ";SE
220 IF SE = 0 THEN 270
230 GOTO 120
240 :
250 REM ALARM
260 :
270 FOR J = 1 TO 100: S = PEEK (-16336): NEXT J
280 END
```

The CALL −868 in line 210 erases the line each time a new number is printed. Try RUNning the program without CALL −868 and see what happens. When prompted in line 70, enter 15.

POKE 33,33

POKE 33,33 is a wonderful tool that you can use in several ways. You can

1. Change the text window width. The normal width is POKE 33,40. You can manipulate the width of the window to create different display effects such as keeping data on the screen while other information is erased or changed. See Chapter 3 for a full explanation.
   Note: If you are unfamiliar with simple editing on the Apple using the ESC I, J, K, and L keys, read THE APPLESOFT TUTORIAL on editing program lines before proceeding.
2. Simplify line editing. We will use lines 10–70 of "THE COUNTDOWN CLOCK" to demonstrate this.

```
10 REM COUNTDOWN CLOCK
20 :
30 :
40 HOME: PRINT TAB( 11)"COUNTD
   OWN CLOCK"
50 :
60 PRINT : PRINT
70 INPUT "ENTER NUMBER OF SECOND
   S: ";SE
```

We can see that lines 40 and 70 split words in strange places to continue the lines. This is because Applesoft uses only 33 of the screen's 40 columns before continuing a program line on the next screen line.

If you do not retype the split words, the PRINT statement will end up with extra spaces. To avoid this, issue POKE 33,33 before LISTing the line to be edited. This adjusts the width of the text window and prevents any spaces from being added. When you use the REPT and right arrow ($- - >$) keys, the cursor will stay within the confines of the new margin. There will be no need to retype part of your program line after editing. When you are through editing, type TEXT to restore the text window to normal.

3. Use it when formatting printed copies. Some printers will PRINT program lines exactly as they appear on the 40-column screen in TEXT mode as in the previous example.

To expand the printed line length to the standard 80 columns, type the following:

```
POKE 33,33
PR#1 (or whatever slot number your printer uses)
LIST
```

When you are finished, type

PR#0 to disconnect the printer.
TEXT to reset the text window to normal.

Lines 10–70 in the previous example now appear as

```
10  REM COUNTDOWN CLOCK
20  :
30  :
40  HOME: PRINT TAB( 11)"COUNTDOWN CLOCK"
50  :
60  PRINT : PRINT
70  INPUT "ENTER NUMBER OF SECONDS: ";SE
```

Each statement now occupies a single line, with no split words.

CTRL-D PROBLEM

If you have ever tried to type a DOS command after GET or a semicolon, you discovered that it doesn't work.

This doesn't work:

```
10 D$ = CHR$ (4): REM CTRL - D
20 PRINT "PRESS AND KEY"
30 GET A$
40 PRINT D$; "CATALOG"
```

This doesn't work either:

```
10 D$ = CHR$ (4): REM CTRL - D
20 PRINT "PRESS AND KEY";
30 PRINT D$; "CATALOG"
```

The semicolon suppressed the RETURN at the end of line 20.

This one works!

```
10 D$ = CHR$ (13) + CHR$ (4): REM CHR$(13)=CARRIAGE RETURN
20 PRINT "PRESS ANY KEY"
30 GET A$
40 PRINT D$; "CATALOG"
```

The carriage return just before CTRL-D [CHR$(4) in line 10] makes the DOS command work. DOS needs a line feed before it executes. Another option is to use INPUT instead of GET.

(Chapter 8)

SIGNAL BELL

Sometimes you want to draw attention to a particular program line or section while LISTing. Some programmers use asterisks or colons in **REM** lines. For example,

```
 90 REM **********
100 REM  VARIABLES USED
110 REM **********
```

or

```
 90 REM ::::::::::
100 REM  VARIABLES USED
110 REM ::::::::::
```

How about making the Apple bell ring when that special section scrolls by? We do this with **CTRL-G.**

You must type CTRL-G as many times as you want the bell to ring. For example, if you want three rings in line 100, do the following:

1. Type **100 REM**
2. Hold the CTRL key down while typing **G.** The bell will ring. Take your fingers off the keys.
3. Repeat step 2 twice.
4. Type **VARIABLES USED.**
5. Press the RETURN key.
6. LIST the program.

The bell rings three times when line 100 appears on the screen.

SECURITY PROCEDURES

There are many ways to prevent people from LISTing or copying your programs. We will discuss several easy but effective techniques.

101
202
WOW! YOU BROKE INTO PUDGY LA BARGES' APPLE IIE IN KENT, WA... YOU'VE GOT HIS GAMES, HIS SOFTWARE AND EVERYTHING!!!
IBN 704
IBN 705

Figure 7-1

POKE Protection

You can protect a program by starting it off with certain POKEs. Here are some POKE values and their meanings.

POKE 214,255 RUNs a disk program when any command is issued. (Normal value is 0.)

POKE 1010,102: POKE 1011,213: POKE 1012,112 RUNs the program again when RESET is pressed.

POKE 1011,0: POKE 1012,0 or POKE 1011,224 Reboots the computer when RESET is pressed.

POKE 50,128 Makes CATALOG and LISTing invisible.

CTRL-C

Many users will try to break or exit a program by using CTRL-C. To prevent this, add

```
 10 ONERR GOTO 1010
  :
  :
  :
1000 REM ONERR ROUTINES
1010 IF PEEK (222) = 255 THEN RUN
```

The error trap will cause any CTRL-C interrupt to reRUN the program.

CTRL Characters

Another type of protection involves using invisible CTRL characters in a file name. Before SAVEing the program, insert an invisible CTRL character somewhere in the file name. For example, in file ANNRPT, you could type

SAVE A**CTRL-E**NNRPT

Do not insert a space between A and CTRL-E or CTRL-E and N. The CTRL-E is a nonprinting character and does not advance the cursor. The filename will appear normal in the CATALOG. If a user tries to RUN ANNRPT, however, the computer displays FILE NOT FOUND. The file can only be accessed by typing the invisible control character exactly as it was SAVEd.

Of course, this type of security could backfire on you if you forget which control character you used and where you put it! But we can solve this problem, too. Read on.

Figure 7-2 War Games.

CTRL Character Revealed

The following program makes all control characters FLASH on the screen.

```
 10 REM CTRL CHARACTERS REVEALED
 20 FOR J = 768 TO 795
 30 READ V: POKE J,V: NEXT J
 40 POKE 54,0: POKE 55,3
 50 CALL 1002
 60 PRINT "ALL CONTROL CHARACTERS WILL NOW APPEAR"
 70 PRINT "AS FLASHING CHARACTERS."
 80 PRINT "TO RECOVER, TYPE 'PR#0'."
 90 :
100 DATA 201,141,240,21,201,136
110 DATA 240,17,201,128,144,13
120 DATA 201,160,176,9,72,132
130 DATA 53,56,233,64,76,249
140 DATA 253,76,240,253
```

Now type CATALOG. If any of the file names has a hidden CTRL character, it will show up in inverse.

Ask Questions

Another simple protection scheme uses an INPUT statement. The user's response must match a predetermined answer exactly, or the program will not RUN.

Example

```
642 INPUT "ENTER SECURITY CODE: ";SC$
644 IF SC$ < > "CODE" THEN NEW
```

See Chapter 3 for another version of this type of security technique.

The Invisible Security Code

One way to hide a security code within the program is to store it in DATA lines in number form. These numbers represent the ASCII decimal equivalents of your secret code. For example,

```
500 DATA 80,65,83,83,87,79,82,68
```

These numbers are the ASCII equivalents of the word PASSWORD. The following short program displays all the ASCII values and the string's length.

```
10 INPUT "ENTER STRING TO BE CODED: ";A$
15 PRINT: PRINT:PRINT "ENTER THESE CODES INTO YOUR
   DATA LINE:"
20 A = LEN (A$): PRINT
30 FOR B = 1 TO A
40 B$ = MID$ (A$,B,1)
50 PRINT ASC (B$);: IF A < > B THEN PRINT ",";
60 NEXT B
70 PRINT:PRINT
80 PRINT "YOUR STRING IS ";A;" CHARACTERS LONG."
```

Add this information to your program as follows:

```
 10 INPUT "ENTER YOUR SECURITY CODE: ";N$
 20 GOSUB 510
 30 :
 40 REM REST OF PROGRAM

500 REM DATA INFORMATION
510 FOR B = 1 TO (put the length of the string here)
520 READ A
530 IF MID$(N$,B,1) < > CHR$(A) THEN GOTO 600
540 NEXT B
550 RESTORE
560 RETURN
570 DATA (put the ASCII codes here)
    :
    :
```

```
600 FOR J = 1 TO 50: S = PEEK (-16336) + PEEK (-16336):
    NEXT J
610 PRINT "ACCESS DENIED. REENTER CODE."
620 PRINT CHR$(4);"RUN"
```

The subroutine beginning at line 500 READs the DATA in line 570. Then each character of the code entered is compared to the character name of the ASCII value. If any discrepancies are found, the program goes to lines 600–620. An alarm sounds, a prompt advises the user ACCESS DENIED, and the program RUNs again. Add further protection by disenabling CTRL-C and the RESET key.

MASTERMIND

MASTERMIND is a popular board game. The following computer version includes instructions on how to play it. This program utilizes several of the techniques we have discussed.

```
 10 REM ::::::::::::::::::::::
 20 :
 30 REM COMPUTER VERSION OF
 40 REM   MASTERMIND
 50 :
 60 REM              BY
 70 REM      WAYNE DYRNESS
 80 REM          (C) 1984
 85 :
 90 REM ::::::::::::::::::::::
100 :
110 :
120 :
130 CLEAR
140 TEXT
150 D$ = CHR$ (4)
160 T = 0 : JJ = 0 : PP$ = ""
190 HOME
200 INVERSE : HTAB 15: PRINT "MASTERMIND": NORMAL
210 VTAB 5
220 Z$ = "< >" + CHR$ (8) + CHR$ (8)
225 :
226 :
230 PRINT "INSTRUCTIONS? <Y><N>: ";Z$;: GET A$: PRINT A$
240 IF A$ = "Y" THEN GOSUB 1790
241 :
242 :
```

```
250 DIM CO$(6)
260 DATA "RED","BLUE","GREEN","YELLOW","WHITE","BLACK"
270 FOR X = 1 TO 6
280 READ CO$(X)
290 NEXT
295 :
300 HOME
310 PRINT "PLEASE WAIT WHILE COMPUTER": PRINT "CREATES A CODE."
320 GOSUB 2300: REM READS IN TONE ROUTINE
330 FOR X = 1 TO 20
340 FOR Z = 1 TO 100: NEXT Z: REM TIME DELAY
350 S = INT ( RND (1) * 200) + 10
360 POKE 0,S; POKE 1,10: CALL 771
370 NEXT X
375 :
376 :
380 REM CREATES RANDOM CODE
381 :
382 :
390 FOR X = 1 TO 4
400 GU(X) = INT (6 * RND (1)) + 1
410 GU$(X) = STR$ (GU(X))
420 NEXT X
425 :
430 REM MAKES SURE EACH NUMBER OF THE CODE IS DIFFERENT
435 :
440 FOR X = 1 TO 3
450 IF GU(X) = GU(X + 1) THEN 390
460 IF GU(X) = GU(X + 2) THEN 390
470 IF GU(X) = GU(X + 3) THEN 390
480 NEXT X
490 :
500 HOME
506 :
510 PRINT "O.K. . . . THE CODE IS READY, FREDDY!"
520 PRINT
530 PRINT "TRY A GUESS . . . "
540 FOR X = 1 TO 1500: NEXT X: REM TIME DELAY
545 :
550 HOME
560 GOSUB 1010: REM LISTS COLORS
570 POKE 34,19: REM SETS TOP MARGIN
580 HOME
590 DIM CL$(15,4) : REM DIMENSIONS 15 MAX GUESSES PER 4 COLORS
600 DIM CL(15,4)
610 FOR N = 1 TO 15: REM ALLOWS 15 GUESSES
620 HOME
```

```
621 :
622 :
630 PRINT "SELECT NUMBER OF FIRST COLOR: ";Z$;
640 GET A$: IF A$ = CHR$ (13) THEN VTAB PEEK (37) + 1:
    CALL - 198: HTAB 1: GOTO 630
650 IF VAL (A$) < 1 OR VAL (A$) > 6 THEN VTAB PEEK (37)
    + 1: CALL - 198: HTAB 1: GOTO 630
660 PRINT A$
670 CL$(N,1) = A$: REM GUESS #1
671 :
672 :
680 PRINT "SELECT NUMBER OF SECOND COLOR: ";Z$;
690 GET A$: IF A$ = CHR$ (13) THEN VTAB PEEK (37) + 1:
    CALL - 198: HTAB 1: GOTO 680
700 IF VAL (A$) < 1 OR VAL (A$) > 6 THEN VTAB PEEK (37)
    + 1: CALL - 198: HTAB 1: GOTO 680
710 PRINT A$
720 CL$(N,2) = A$: REM GUESS #2
721 :
722 :
730 PRINT "SELECT NUMBER OF THIRD COLOR: ";Z$;
740 GET A$: IF A$ = CHR$ (13) THEN VTAB PEEK (37) + 1:
    CALL - 198: HTAB 1: GOTO 730
750 IF VAL (A$) < 1 OR VAL (A$) > 6 THEN VTAB PEEK (37)
    + 1: CALL - 198: HTAB 1: GOTO 730
760 PRINT A$
770 CL$(N,3) = A$: REM GUESS #3
771 :
772 :
780 PRINT "SELECT NUMBER OF FOURTH COLOR: ";Z$;
790 GET A$: IF A$ = CHR$ (13) THEN VTAB PEEK (37) + 1:
    CALL - 198: HTAB 1: GOTO 780
800 IF VAL (A$) < 1 OR VAL (A$) > 6 THEN VTAB PEEK (37)
    + 1: CALL - 198: HTAB 1: GOTO 780
810 PRINT A$
820 CL$(N,4) = A$: REM GUESS #4
821 :
822 :
830 GOSUB 1070: REM GOES TO COMPARE GUESS WITH ANSWER
835 :
840 NEXT N: REM GOES TO ENTER NEXT GUESS
845 :
850 POKE 34,0: REM RESET TOP MARGIN
851 :
853 :
860 REM : : : : : : : : : : : : : : : :
861 :
```

```
 862 REM      LOSER!
 863 :
 864 REM : : : : : : : : : : : : : : : :
 865 :
 866 :
 870 HOME
 880 VTAB 10
 890 CALL - 868: INVERSE : HTAB 15: PRINT "YOU LOSE!": NORMAL
 900 :
 910 FOR X = 1 TO 100
 920 SOUND = PEEK ( - 16336): REM BUZZ
 930 NEXT X
 940 PRINT : PRINT : PRINT "THE CORRECT ANSWER IS: "
 950 FOR X = 1 TO 4
 960 PRINT
 970 PRINT GU(X);"   ";CO$(GU(X))
 980 NEXT X
 982 :
 983 :
 990 GOTO 1660: REM GOES TO GIVE ANOTHER CHANCE TO PLAY
 992 :
 993 :
1000 REM SUB FOR LISTING COLORS
1010 FOR X = 1 TO 6
1020 PRINT "<";X;"> - ";CO$(X)
1030 NEXT X
1040 RETURN
1042 :
1043 :
1060 REM SUB TO COMPARE GUESSES WITH CORRECT ANSWERS
1065 :
1070 FOR X = 1 TO 4:GG$(X) = GU$(X): NEXT : REM GG$(X) =
     TEMPORARY COMPUTER CODE
1075 :
1076 :
1080 FOR X = 1 TO 4
1085 :
1090 FOR D = 1 TO 4
1095 :
1100 REM EACH COLOR GUESS YOU MAKE IS COMPARED WITH EACH
     OF THE FOUR CODES THE COMPUTER CREATED.
1105 :
1110 IF CL$(N,X) < > GG$(D) THEN 1170: REM NO MATCH-GOES
     TO NEXT ? #
1120 MT$(X) = "0": REM COLOR MATCH FOUND
1130 IF X = D THEN MT$(X) = "X": REM POSITION MATCH FOUND
```

```
1140 GG$(D) = "0": REM NO LONGER LETS THIS VALUE BE
     CONSIDERED A POSSIBILITY
1145 :
1150 GOTO 1190: REM GOES TO COMPARE YOUR NEXT GUESS
     WITH THE COMPUTER'S CODE
1160 :
1170 NEXT D
1175 :
1180 MT$(X) = " ": REM NO COLOR OR POSITION MATCH FOUND.
     SETS UP TO GIVE CLUE.
1185 :
1190 NEXT X
1195 :
1200 :
1210 REM SUB GOES TO CHECK WHETHER ORDER IS SAME
1212 :
1215 GOSUB 1480
1216 :
1220 PRINT
1222 :
1223 :
1224 REM ::::::::::::::::::::::
1225 :
1226 REM  COMPUTER GIVES CLUES
1227 :
1228 REM ::::::::::::::::::::::
1230 T = T + 1
1250 POKE 33,20: POKE 32,20: REM RESET WIDTH AND LEFT MARGIN
1260 HOME
1270 VTAB T
1275 :
1280 REM PRINTS CLUE OUT OF ORDER TO AVOID GIVING AWAY ANSWER
1285 :
1290 IF MT$(2) = " " THEN V = V + 1: GOTO 1310
1300 PRINT MT$(2);
1310 IF MT$(4) = " " THEN V = V + 1: GOTO 1330
1320 PRINT MT$(4);
1330 IF MT$(1) = " " THEN V = V + 1: GOTO 1350
1340 PRINT MT$(1);
1350 IF MT$(3) = " " THEN V = V + 1: GOTO 1370
1360 PRINT MT$(3);
1370 IF V = 0 THEN 1390
1380 FOR E = 1 TO V: PRINT " ";: NEXT E
1390 PRINT " ";
1395 :
1400 FOR X = 1 TO 4
```

```
1410 PRINT CL$(N,X);
1420 NEXT X
1425 :
1430 POKE 33,40: POKE 32,0: REM RESET LEFT MARGIN AND WIDTH
1435 :
1440 JJ = 0
1450 V = 0
1460 RETURN
1461 :
1462 :
1463 REM : : : : : : : : : : : : : : : : : : :
1464 :
1465 REM      WINNER!!!
1466 :
1467 REM : : : : : : : : : : : : : : : : : : :
1470 :
1480 FOR X = 1 TO 4
1490 IF CL$(N,X) < > GU$(X) THEN RETURN
1500 NEXT X
1505 :
1510 HOME
1520 VTAB 10: CALL - 958: PRINT : PRINT
1530 FLASH : HTAB 15
1540 PRINT "YOU WON ! ! !"
1550 NORMAL
1555 :
1560 FOR X = 1 TO 10
1570 CALL - 198: REM RING BELL
1580 NEXT X: PRINT : PRINT
1585 :
1590 PRINT "YOUR CORRECT ANSWER WAS: "
1600 PRINT
1605 :
1610 FOR X = 1 TO 4: PRINT GU$(X); SPC( 5);: NEXT X : PRINT
1620 FOR X = 1 TO 4
1630 PRINT CO$(GU(X));" ";
1640 NEXT X
1645 :
1650 PRINT
1651 :
1652 :
1653 REM : : : : : : : : : : : : : : : : : : : : : :
1654 :
1655 REM  CHANCE TO PLAY AGAIN
1656 :
1657 REM : : : : : : : : : : : : : : : : : : : : : :
1660 VTAB 23
1670 PRINT "PLAY AGAIN? <Y> <N>: ";Z$;: GET A$: IF A$ = "Y" THEN 10
```

```
1680 TEXT
1690 HOME
1700 FOR X = 1 TO 50
1710 S = PEEK ( - 16336):SO = PEEK (S): REM SOUND ROUTINE
1720 NEXT X
1722 :
1723 :
1724 REM : : : : : : : : : : : : : : : : : : : : : : :
1725 :
1726 REM      PARTING WORDS
1727 :
1728 REM : : : : : : : : : : : : : : : : : : : : : : :
1729 :
1730 PRINT "WHAT'S THE MATTER?": PRINT : PRINT : PRINT
     "COULDN'T HANDLE THE PRESSURE?"
1740 FOR X = 1 TO 1500: NEXT X: REM TIME DELAY LOOP
1750 VTAB 22: HTAB 15
1760 PRINT "BYE";: CALL - 198: PRINT "-BYE": CALL - 198
1770 END
1771 :
1772 :
1773 REM : : : : : : : : : : : : : : : : : : : : : : :
1774 :
1775 REM      INSTRUCTIONS
1776 :
1777 REM : : : : : : : : : : : : : : : : : : : : : : :
1778 :
1780 :
1790 HOME
1800 PRINT " MASTERMIND IS AN EXCITING GAME OF": PRINT
     "STRATEGY AND A CHALLENGE FOR THE MIND."
1810 PRINT
1820 PRINT " THE COMPUTER WILL, AT RANDOM, SELECT"
1830 PRINT "FOUR DIFFERENT COLORS. YOUR"
1840 PRINT "CHALLENGE IS TO DETERMINE WHAT THOSE FOUR"
1850 PRINT "COLORS ARE. YOU ARE GIVEN SIX COLORS"
1860 PRINT "TO CHOOSE FROM."
1870 PRINT
1880 PRINT " THE COMPUTER WILL EVALUATE YOUR "
1890 PRINT "GUESSES. IT WILL GIVE YOU CLUES TO"
1900 PRINT "LET YOU KNOW IF YOU ARE CLOSE TO ": PRINT "GUESSING
     THE CORRECT COLORS AS": PRINT "WELL AS THE EXACT ORDER."
1910 PRINT
1920 VTAB 23
1930 PRINT "PRESS ANY KEY TO CONTINUE. ";Z$;: GET A$
1940 HOME
1950 PRINT "THE COMPUTER WILL RESPOND TO YOUR"
1960 PRINT "GUESS IN THREE WAYS: "
```

```
1970 PRINT : PRINT
1980 PRINT " 0 = CORRECT COLOR"
1990 PRINT "X = CORRECT COLOR & POSITION"
2000 PRINT
2010 PRINT "   = NOTHING INDICATES YOU HAVE NOT"
2020 PRINT "       GUESSED EITHER COLOR OR "
2030 PRINT "       POSITION."
2040 PRINT : PRINT
2050 PRINT " OF COURSE, THE 'X' OR '0' ARE IN NO "
2060 PRINT "PARTICULAR ORDER AND THEIR POSITIONS"
2070 PRINT "MAY NOT CORRESPOND TO THE ORDER"
2080 PRINT "OF YOUR COLOR CHOICES."
2100 VTAB 23
2110 PRINT "PRESS ANY KEY TO CONTINUE. ";Z$;: GET A$
2120 HOME
2130 PRINT " YOU HAVE 15 TRIES TO GUESS NOT"
2140 PRINT "ONLY THE CORRECT COLORS THAT THE COM-"
2150 PRINT "PUTER HAS SELECTED. BUT ALSO THE EXACT"
2160 PRINT "ORDER IN WHICH THEY APPEAR."
2170 PRINT
2180 PRINT " REMEMBER! ALL FOUR COLORS": PRINT "SELECTED BY
     THE COMPUTER WILL BE"
2190 PRINT "DIFFERENT. THERE WILL BE NO": PRINT "DUPLICATES."
2200 PRINT : PRINT
2210 PRINT "IF YOU ARE READY TO BEGIN THIS EXCITING"
2220 PRINT "GAME, PRESS <RETURN>."
2230 PRINT : PRINT
2240 PRINT " . . . . GOOD LUCK."
2250 VTAB 23
2260 PRINT "- - - - > ";Z$;: GET A$
2270 PRINT
2280 RETURN
2281 :
2282 :
2283 REM : : : : : : : : : : : : : : : : : : : : : : :
2284 :
2285 REM   TONE ROUTINE SUB
2286 :
2287 REM : : : : : : : : : : : : : : : : : : : : : : :
2290 :
2300 DATA 173,48,192,136,208,4,198,1,240,8,202,208,246,166,
          0,76,3,3,96
2310 FOR J = 771 TO 789: READ X: POKE I, X: NEXT
2320 RETURN
```

You could enhance the program by adding other sound or tone routines.

RANDOM NUMBERS

RND is a **RaNDom** number generator. It appears in a program as **RND (N),** where N is equal to a number. Although RND isn't perfect, its results are adequate for most purposes.

If the argument is greater than zero, RND generates a new random number each time. The number will always be decimals between zero and one, but never exactly one. Let's see how this works.

```
10 FOR J = 1 TO 10
20 PRINT RND(1)
30 NEXT J
40 END
```

This program produces 10 RND numbers between zero and one. But how many times do you need random numbers between zero and one, especially decimal??!!! Maybe never.

To produce whole numbers, use

```
10 FOR J = 1 TO 10
20 LET X = INT (RND(1) * 40)
30 PRINT X
40 NEXT J
50 END
```

Line 10 determines how many numbers are to be printed. Line 20 produces a random number between zero and one. That value is multiplied by 40. The **INT** (integer) function eliminates any decimal fraction, leaving just the whole number. The result is 10 numbers between 1 and 40. Line 30 prints the numbers. By changing the argument "40" in line 20, you can produce any random number between one and the argument.

The above program produces numbers between 1 and 40, but not 40. To include it, change line 20 to

```
20 LET X = INT(RND(1) * 40) + 1
```

If the argument is equal to zero, RND won't generate a new number. It uses the same number it generated previously.

```
10 FOR J = 1 TO 10
20 PRINT RND(0)
30 NEXT J
40 END
```

This program generates the same random number 10 times.

Figure 7-3

If the argument is less than zero, i.e., a negative number, RND produces the same sequence of numbers every time; for example,

```
10 LET X = RND(-5)
20 FOR J = 1 TO 10
30 PRINT RND (1)
40 NEXT J
50 END
```

Suppose we change the argument in line 10 to −8, RND will produce a different sequence of numbers. But as before, the program will produce this new sequence every time you RUN the program.

HANDY SUBROUTINES

Type this program and SAVE it on disk. Use the MERGE function of the RE-NUMBER program to APPEND it to the end of your file.

```
2000 REM PRINTS HYPHENATED LINE
2010 FOR J = 1 TO 39: PRINT "-";:NEXT J: PRINT "-": RETURN

2030 REM PRINT ASTERISK LINE
2040 FOR J = 1 TO 39: PRINT "*";:NEXT J: PRINT "*": RETURN
```

```
2050 REM RETURN TO CONTINUE
2060 INVERSE:PRINT "PRESS RETURN TO CONTINUE":NORMAL:RETURN

2080 REM RING BELL ONCE
2090 CALL -198: RETURN

2100 REM RING BELL THREE TIMES
2110 PRINT "": RETURN: REM USE CTRL-G IN QUOTES 3 TIMES

2120 REM OUTPUT TO PRINTER
2130 PRINT D$; "PR#1":RETURN

2140 REM DISCONNECT PRINTER
2150 PRINT D$;"PR#0":RETURN
```

CALENDAR

This program prints a calendar for any year from 1591 to the year of your choice.

```
 100 GOTO 30010
 200 REM ** SET WINDOW **
 210 POKE 32,1: POKE 33,37: POKE 34,1: POKE 35,16: HOME : RETURN
3000 REM *** CHOICE OF MONTH ***
3004 GOSUB 210
3006 VTAB 7: HTAB 4: INPUT "ENTER YEAR: (YYYY)? ";Y
3007 IF Y < = 1581 THEN CALL - 198: VTAB PEEK (37): GOTO 3006
3010 VTAB 8: HTAB 4: CALL - 958: INPUT "ENTER MONTH: (MM)? ";M
3014 IF M > 0 AND M < 13 THEN 3990
3020 HTAB 4: PRINT "PARDON?"; CHR$ (7): FOR X = 1 TO 1000: NEXT X:
     GOTO 3010
3990 RETURN
5000 REM *** DISPLAY MONTH ***
5010 GOSUB 200
5020 YY = INT (Y - 1):MM = INT (M + 10): IF MM < 13 THEN 5040
5030 MM = MM - 12:YY = YY + 1
5040 DD = INT (D0 + F0(MM) + INT (YY / 4) + INT (YY / 400) -
     INT (YY / 100) + YY)
5050 DD = INT (1.0001 + DD - 7 * INT (DD / 7))
5060 VTAB 4: HTAB 17 - LEN (MO$(MM)) / 2
5070 PRINT MO$(MM);", ";Y
5080 VTAB 8: HTAB 6
5090 PRINT "SUN MON TUE WED THR FRI SAT"
5100 IF MM < 12 THEN 5200
5110 LAST = 28
5120 IF INT (Y - 4 * INT (Y / 4)) = 0 AND INT (Y - 100 *
     INT (Y / 1000)) < > 0 THEN LAST = 29
```

```
5130 IF INT (Y - 400 * INT (Y / 400)) = 0 THEN LAST = 29
5140 GOTO 5300
5200 LAST = F0(MM + 1) - F0(MM)
5300 FIRST = DD:D1 = 1
5310 FOR WEEK = 1 TO 6: VTAB 9 + WEEK: FOR DAY = 1 TO 7: HTAB 4
     * DAY + 2
5320 IF D1 = 1 AND DAY < FIRST THEN 5360
5330 IF D1 > LAST THEN 5360
5340 IF D1 < 10 THEN PRINT CHR$ (32);
5350 PRINT D1;:D1 = D1 + 1
5360 NEXT DAY : NEXT WEEK
5370 TEXT
5990 RETURN
6000 REM **** BUILD 3 MONTHS ****
6001 REM USED BY PRINTER
6002 REM
6010 FOR I = 1 TO 7:LINE$(I) = "": FOR J = 1 TO 80:LINE$(I) =
     LINE$(I) + CHR$ (32): NEXT J:X = FRE (0): NEXT I
6020 FOR M1 = 1 TO 3:M = 3 * A + M1
6030 YY = INT (Y - 1):MM = INT (M + 10): IF MM < 13 THEN 6050
6040 MM = MM - 12:YY = YY + 1
6050 DD = INT (D0 + F0(MM) + INT (YY / 4) + INT (YY / 400) -
     INT (YY / 100) + YY)
6060 DD = INT (1.0001 + DD - 7 * INT (DD / 7))
6070 LINE$(7) = LEFT$ (LINE$(7),T(M1)) + MO$(MM) + RIGHT$
     (LINE$(7), LEN (LINE$(7)) - T(M1) - LEN(MO$(MM)))
6080 IF MM < 12 THEN 6120
6090 LAST = 28: IF INT (Y - 4 * INT (Y / 4)) = 0 AND INT (Y -
     100 * INT (Y / 100)) < > 0 THEN LAST = 29
6110 GOTO 6200
6120 LAST = F0(MM + 1) - F0(MM)
6200 FIRST = DD:D1 = 1
6210 FOR WEEK = 1 TO 6: FOR DAY = 1 TO 7
6220 IF D1 = 1 AND DAY < FIRST THEN 6290
6230 IF D1 > LAST THEN 6290
6240 A$ = STR$ (D1)
6250 LINE$(WEEK) = LEFT$ (LINE$(WEEK),T(M1) + 3 * DAY - LEN
     (A$)) + A$ + RIGHT$ (LINE$(WEEK), LEN (LINE$(WEEK)) - T(M1)
     - 3 * DAY)
6260 D1 = D1 + 1
6290 NEXT DAY
6292 NEXT WEEK
6294 NEXT M1
6990 RETURN
21000 REM **** SCREEN MENU ****
21010 TEXT : HOME
21020 INVERSE : VTAB 1: HTAB 1: FOR 1 = 1 TO 39: PRINT CHR$
     (32);: NEXT I
```

```
21030 FOR I = 1 TO 18: VTAB I; HTAB 1: PRINT CHR$ (32);: HTAB
      39: PRINT CHR$ (32);: NEXT I
21040 VTAB 18: HTAB 1: FOR I = 1 TO 39: PRINT CHR$ (32);: NEXT I
21050 NORMAL
21060 GOSUB 3004: REM CHOICE
21070 GOSUB 5010: REM DISPLAY MONTH
21080 TEXT : VTAB 20:T = 8
21090 HTAB T: PRINT " <LEFT> . . . LAST MONTH"
21100 HTAB T: PRINT "<RIGHT> . . . NEXT MONTH"
21110 HTAB T: PRINT "<SPACE> . . . ANOTHER CHOICE"
21120 HTAB T: PRINT "  <RTN> . . . RETURN TO MENU"
21150 VTAB 23: HTAB 1: GET A$
21160 IF A$ < > CHR$ (8) THEN 21180
21170 M = M - 1: IF M > = 1 THEN 21174
21172 M = 12:Y = Y - 1
21174 GOSUB 5010: GOTO 21150
21180 IF A$ < > CHR$ (21) THEN 21190
21182 M = M + 1: IF M < 13 THEN 21186
21184 M = 1:Y = Y + 1
21186 GOSUB 5010: GOTO 21150
21190 IF A$ < > CHR$ (32) THEN 21210
21200 GOSUB 3004: REM CHOICE
21202 GOSUB 5010: REM DISPLAY MONTH
21204 GOTO 21150
21210 IF A$ = CHR$ (13) THEN 21900
21220 GOTO 21150: REM IGNORE
21900 RETURN
22000 REM **** PRINTER MENU ****
22010 TEXT: HOME
22015 VTAB 6
22020 INPUT "PRINTER SLOT? (1-4) ";SLOT
22030 IF SLOT < 1 OR SLOT > 4 THEN PRINT CHR$ (7); "PARDON?":
      GOTO 22020
22040 PRINT : PRINT
22050 INPUT "WHICH YEAR?(YYYY): ";Y0$
22052 IF LEN (Y0$) < > 4 THEN 22050
22054 Y1 = VAL ( LEFT$ (Y0$,1)):Y2 = VAL ( MID$ (Y0$,2,1)):Y3
      = VAL ( MID$ (Y0$,3,1)):Y4: = VAL ( RIGHT$ (Y0$,1))
22056 Y = VAL (Y0$)
22060 IF Y < 1582 OR Y > 2999 THEN PRINT CHR$ (7); "PARDON?":
      GOTO 22050
22062 PRINT
22064 INPUT "HOW MANY COPIES? (1-20): ";NC%: IF NC% < 1 OR NC% >
      20 THEN 22062
22066 PRINT : PRINT
22070 PRINT "START UP THE PRINTER. THEN"
22080 PRINT "PRESS A KEY  ";: GET A$
22090 PRINT : PRINT : HTAB 15: PRINT " . . . STANDBY . . . "
```

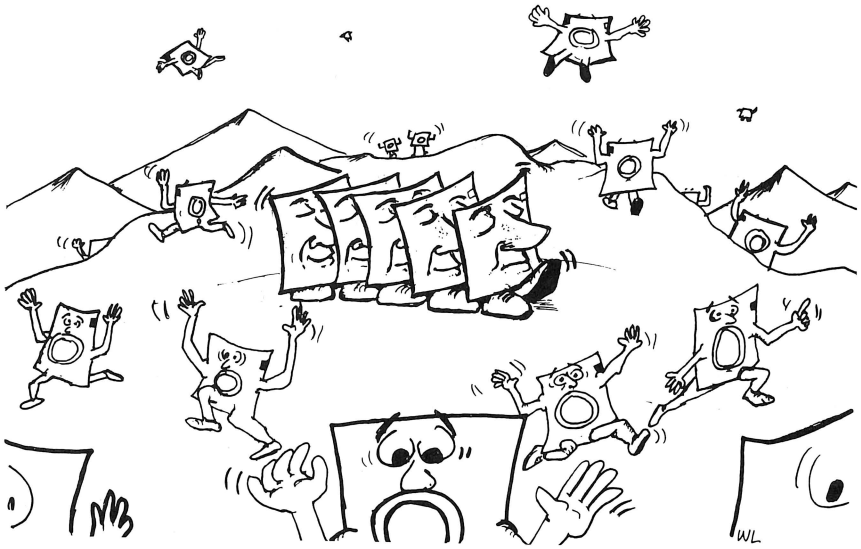
```
22100 PRINT : PRINT : PRINT
22200 PR# SLOT
22201 PRINT CHR$ (9); "80N"
22210 FOR CN = 1 TO NC%
22220 PRINT CHR$ (32)
22222 PRINT CHR$ (32)
22224 FOR L = 1 TO 7
22226 FOR I = 1 TO 20: PRINT CHR$ (32);: NEXT I
22228 PRINT Y$(L,Y1 + 1);"     ";Y$(L,Y2 + 1);"     "; Y$(L,Y3 + 1);
      " ";Y$ (L,Y4 + 1)
22230 NEXT L
22232 PRINT CHR$ (32): PRINT CHR$ (32)
22234 PRINT CHR$ (32)
22240 FOR A = 0 to 3
22250 GOSUB 6010
22260 PRINT LINE$(7)
22262 A$ = "S M T W T F S"
22264 PRINT "    ";A$;"          ";A$;"          ";A$
22270 FOR I = 1 TO 6: PRINT LINE$(I): NEXT I
22280 FOR I = 1 TO 3: PRINT CHR$ (32): NEXT I
22290 NEXT A
22291 PRINT "<<< HAVE A GREAT YEAR! >>>"
22300 FOR I = 1 TO 8: PRINT CHR$ (32): NEXT I
22400 NEXT CN
22500 PR# 0
22990 RETURN
30000 REM **** CONSTANTS ****
30010 D0 = 3: REM PHASE DAY OF WEEK
30050 REM START THE YEAR ON MARCH FIRST AND TABLE THE
30052 REM FIRST DAY OF EACH MONTH. THIS WORKS WHEN
30054 REM FEBRUARY IS THE LAST MONTH OF THE YEAR!
30100 DIM F0(12): DATA 0,31,61,92,122,153,184,214,245,275,306,337:
      FOR I = 1 TO 12: READ F0(I): NEXT I
30120 DIM MO$(12): DATA MARCH,APRIL,MAY,JUNE,JULY,AUGUST,SEPTEMBER,
      OCTOBER,NOVEMBER,DECEMBER
30130 DATA JANUARY,FEBRUARY: FOR I = 1 TO 12: READ MO$(I): NEXT I
30140 DIM LINE$(7)
30150 DIM T(3):T(1) = 1:T(2) = 28:T(3) = 55
30200 REM ZERO
30201 DATA " **** "
30202 DATA "**  **"
30203 DATA "**  **"
30204 DATA "**  **"
30205 DATA "**  **"
30206 DATA "**  **"
30207 DATA " **** "
30210 REM ONE
```

```
30211 DATA "  **  "
30212 DATA " *** "
30213 DATA "  **  "
30214 DATA "  **  "
30215 DATA "  **  "
30216 DATA "  **  "
30217 DATA " **** "
30220 REM TWO
30221 DATA " **** "
30222 DATA " **  ** "
30223 DATA " **  ** "
30224 DATA "    ** "
30225 DATA " **    "
30226 DATA " ** "
30227 DATA " ****** "
30230 REM THREE
30231 DATA " **** "
30232 DATA "**  **"
30233 DATA "    ** "
30234 DATA "   **   "
30235 DATA "    ** "
30236 DATA " **  ** "
30237 DATA " **** "
30240 REM FOUR
30241 DATA "   ** "
30242 DATA "  *** "
30243 DATA " *  ** "
30244 DATA "******"
30245 DATA "   ** "
30246 DATA "   ** "
30247 DATA "   ** "
30250 REM FIVE
30251 DATA "******"
30252 DATA " **     "
30253 DATA " **     "
30254 DATA "***** "
30255 DATA "    ** "
30256 DATA "** **"
30257 DATA " **** "
30260 REM SIX
30261 DATA " **** "
30262 DATA "** **"
30263 DATA " **     "
30264 DATA "***** "
30265 DATA "** **"
30266 DATA "** **"
30267 DATA " **** "
```

```
30270 REM SEVEN
30271 DATA " ****** "
30272 DATA " *     ** "
30273 DATA "   ** "
30274 DATA "     **     "
30275 DATA " **        "
30276 DATA "**     "
30277 DATA "**     "
30280 REM EIGHT
30281 DATA " **** "
30282 DATA " **   ** "
30283 DATA " **   ** "
30284 DATA " **** "
30285 DATA "**   **"
30286 DATA "**   **"
30287 DATA " **** "
30290 REM NINE
30291 DATA " **** "
30292 DATA "**   **"
30293 DATA "**   **"
30294 DATA " **** "
30295 DATA "   **   "
30296 DATA " **    "
30297 DATA "**       "
30300 DIM Y$(7,10)
30310 FOR I = 1 TO 10
30320 FOR J = 1 TO 7: READ Y$(J,I): NEXT J
30330 NEXT I
30990 REM
30992 REM
31000 REM **** MAINLINE MENU ****
31010 TEXT : HOME : NORMAL : SPEED= 255
31020 VTAB 3: HTAB 16: PRINT "CALENDAR"
31022 PRINT
31026 PRINT : HTAB 9: PRINT "WRITTEN BY PAUL IRWIN"
31030 T = 12: VTAB 10
31040 HTAB 5: PRINT "A PERPETUAL CALENDAR PROGRAM FOR"
31050 PRINT "YOUR CHOICE OF SCREEN OR PRINTER:"
31060 PRINT : HTAB T: PRINT "S . . . SCREEN"
31070 PRINT : HTAB T: PRINT "P . . . PRINTER"
31080 PRINT : HTAB T - 4: PRINT "<RTN> . . . RETURN TO BASIC"
31120 X = FRE (0)
31130 VTAB 20: HTAB 16: PRINT "WHICH? ";: GET A$
31140 IF A$ < > "S" THEN 31160
31150 GOSUB 21000: GOTO 31010
31160 IF A$ < > "P" THEN 31180
31170 GOSUB 22000: GOTO 31010
```

```
31180  IF A$ < > CHR$ (13) THEN 31130
32000  TEXT : HOME : VTAB 12: HTAB 9
32010  PRINT "CALENDAR IS OFF THE AIR"
32020  VTAB 23
32767  END
```

Extracted from *Public Domain Software*, by Paul Irwin.



8 Introduction to Random-Access and Sequential Text Files

As your programs become larger, you eventually will need to create a system to store and retrieve information.

For years, most business offices have kept files in metal cabinets. File clerks spend hours pulling folders and putting them back in the drawers. This form of information management is costly and requires a lot of space.

Today, with the use of computers in the office or even the home, information management can be more efficient. Hundreds of pages of text, numbers, calculations, drafts, forms, etc. can all be stored on one floppy disk and later retrieved within a matter of seconds.

This information (or *data*) is stored by the computer in a special file called the *data* or *text file*. When you CATALOG a disk, you will see something similar to the following:

```
*A 002 HELLO
*A 013 RADISH DESSERTS
*B 021 SPACE MUFFIN
*I 002 APPLESOFT SAUCE
*A 041 BILLING PROGRAM
```

```
T 014 MONTH 1
T 003 MONTH 2
```

The letters in the left-hand column indicate whether the program is (A) a BASIC program file, (B) a binary program file (machine language), (I) an integer BASIC program file, or (T) a data or text file.

The numbers in the next column indicate the number of sectors the program uses on a disk. The last column is the program file name. It must begin with a letter and cannot exceed 30 characters.

The asterisk (*) tells you if a file is locked. Locking a file protects against accidental erasure or overwriting a program you want to keep.

A program file and a data file are different. A program file contains a BASIC program. You can use such commands as RUN, LOAD, LIST, and SAVE with program files. These commands cannot be used with a text file. A text file can be accessed only through or within a BASIC program. A text file is only a storage device, much like a filing cabinet.

Typically, text files are telephone lists, mailing lists, billing and accounting information, inventory, form letters, manuscripts, and so on. This book, in fact, was written on the Apple computer, and each chapter was stored in a text file.

SEQUENTIAL AND RANDOM-ACCESS TEXT FILES

We will discuss the two types of text files, sequential and random-access. Each has its advantages and disadvantages, which we will discuss in greater detail later.

But first, let's quickly review some important factors that will aid us in writing more readable and logical programs.

Modular Programming

Organization in a program is of utmost importance. Not only is a well-organized program easier to read, it is also easier to "debug" (identify and correct errors). See Chapter 6 for additional information on debugging.

Organization means grouping particular functions together in *modules,* hence the term *modular programming.*

When organizing a program, you may structure it as follows:

DATA ENTRY
DATA ANALYSIS
DATA MANIPULATION OR COMPUTATION
FILE UPDATING
EDITING
REPORT GENERATION

Each function requires a particular set of instructions. Each set of instructions belongs in its own module.

Some tasks must be repeated several times during a program. A subroutine module is ideal for this situation.

Clarity

Another important factor is program clarity. You should always document a program with **REM** (**REM**ark) statements.

An introduction module should list the program's name, author, date, program instructions, variables, arrays, files used, etc., as in the following example.

```
 10 REM ********************
 20 REM  BILL PRINT PROGRAM
 30 REM ********************
 40 :
 45 :
 50 REM BY WAYNE DYRNESS
 60 REM (C) 1983
 70 :
 80 REM THIS PROGRAM ALLOWS USER TO INPUT INVOICING AND BILLING
    INFORMATION. PROGRAM WILL AUTOMATICALLY COMPUTE EXTENDED
 90 REM PRICE AND TAX ON RETAIL, WHOLESALE, AND NONTAXABLE ITEMS.
100 REM IT ALSO PRINTS DATA TO PAPER AND STORES IT IN TEXT FILES.
110 :
120 REM **** IMPORTANT: ***** PROGRAM IS SET UP TO TAKE ONLY
    MAX OF 6 DIFFERENT PRODUCT DESCRIPTIONS AND MAX. OF 3
    MISC. ITEMS PER INVOICE.
130 :
140 :
150 :
170 REM ********************
180 REM    VARIABLES USED
190 REM ********************
200 :
210 REM R1 = REC COUNT
220 REM R2 = INV # COUNT
230 REM B1-B5 = ADD & MULTIPLY
240 REM BB= TAB
250 :
260 REM S1 = SALES TAX %
270 :
280 REM V:Y = PRODUCT COUNT
290 REM X = FOR NEXT LOOPS
300 REM Z = DATE (6)
```

```
310 :
320 REM C1$(4) = CHECK #
330 REM CK$(1)= 0=CASH, 1=CHECK, 2=CHARGE
340 :
350 REM N$ = CUSTOMER NAME (15)
360 REM AD$ = ADDRESS (25)
370 REM CT$ = CITY (15)
380 REM ZP$ = ZIP CODE (5)
385 REM ST$ = STATE (2)
390 REM R: R$ = RTL/WHSL (1)
400 REM P$:H$ = PROD. DESCRPTN. (20)
410 REM SP$:P = SELLING PRICE (8)
420 REM QS$:S = QTY SOLD (5)
440 REM MS$: I$ = MISC. ITEM DESCRPTN. (20)
450 REM T$:T = MISC AMT CHRG (8)
460 REM D$ = CTRL-D: TU$ = CTRL-H
465 REM CS$=CLEAR SCREEN IN 80 COLUMN MODE
470 REM R2=INVOICE # (5)
480 REM TL$="0" RETAIL TL$="1" WHOLESALE INSTATE TL$="2"
    WHOLESALE OUT-STATE
490 REM V:Y=VARIABLES IN FOR NEXT LOOP, EACH USE (2) BYTES
500 :
510 :
520 REM ************
530 REM  FILES USED
540 REM ************
550 :
560 :
570 :
580 REM FILE NAME: INV #
590 :
600 REM THIS FILE AUTOMATICALLY PROVIDES AN INVOICE #
610 :
620 REM FILE LENGTH: 5
630 REM DATASET FORMAT: R2
640 :
650 REM RA FILE NAME: BILLING
660 REM FILE LENGTH: 400
670 REM DATASET SET FOR MAX. OF 6 ITEMS AND 3 MISC. ITEMS
680 :
690 :
700 :
```

Including all this information may seem like a waste of time and space. It is, however, an invaluable resource, particularly in a large program. It explains the program and helps avoid duplication of variables.

Each module should include a REM to identify it, for example,

```
500 REM TYPEWRITER SUBROUTINE
505 :
510 FOR J = 1 TO LEN (S$)
520 :
530 etc.
```

Note that a REM or a colon (:) can stand alone on a program line. The colon can enhance readability by placing empty lines between sections.

Error Checking

The integrity of a file depends on the correctness of the data you enter. If you fail to build error traps into the program during the data entry stage, the text files may not be accurate. This could stop the program unexpectedly.

If you are unfamiliar with error-trapping methods or if you need a refresher course, read Chapter 6.

Variable Names

The computer lets you use multicharacter variable names such as NAME$, GUESS, ADDRESS$, QUANTITY, etc. However, it recognizes only the first two characters.

For example, the computer recognizes only NA$ in NAME$. Thus it would identify SALE and SALARY as the same variable (i.e., *SA*le and *SA*lary).

You must also avoid using variables that Applesoft reserves for special purposes, such as FOR, AND, VAL, or DIM (see Appendix F).

Try to limit all variables to two characters. Remember, identifying variables in the introduction module helps avoid duplication.

Variables can include numbers, but they must begin with a letter. Examples are

A	D9$
A1	AZ
D$	AZ$

String Variables

Your computer can do calculations, as in the following program.

```
10 A = 1: B = 2
20 C = A + B
30 PRINT C
40 END
```

Line 10 assigns the value 1 to variable A, and 2 to B. Line 20 makes the variable C = A + B; that is, C = 1 + 2. Line 30 prints 3, the new value of C.

The computer can just as easily handle a string of characters at a time. A string variable can contain any combination of *alphanumeric characters* (letters, numbers, or symbols) and can be up to 255 characters long. A string without any characters is called a "null" string, as in

```
A$ = ""
```

There are no spaces between the quotation marks.

We identify a string variable by the dollar sign at the end, as in

```
10 NM$ = "JOHN"
```

The characters of a string variable must be in quotation marks (e.g., "JOHN"). Typing

```
10 NM$ = "JOHN"
20 PRINT NM$
```

prints JOHN without the quotation marks.

The variable A and the string variable A$ are different. If A = 3 and A$ = "3", the computer would treat them differently. For example,

```
10 A = 3 : A$ = "3"
20 LET C = A + A$
30 PRINT C
40 END
```

produces

```
?TYPE MISMATCH ERROR IN LINE 20
```

The computer treats the "3" in line 10 as a letter or nonnumeric character because it is inside quotation marks. You cannot add 3 to a letter.

You can, however, use the VAL command. It ". . . attempts to interpret a string, up to the first nonnumeric character, as a real or an integer, and returns the value of that number."[1] The following example shows how this works.

```
10 A = 1: B$ = "3"
20 LET B = VAL (B$)
30 C = A + B
40 PRINT C
50 END
```

Line 20 makes the variable B equal to the number within the string. Now we can add A and B together and make the sum equal to C.

Or we could write

```
10 A = 1: B$ = "3"
20 LET C = A + VAL (B$)
30 PRINT C
40 END
```

If B$ = "345BET" the VAL (B$) would be equal to 345.

By now, you may be asking yourself,

"Why make B$ = "3" and then change it to B = VAL (B$)? Why not forget about using the string B$ altogether and just write B = 3?"

The benefit of string variables is that we can manipulate them very easily. Such commands as RIGHT$, LEFT$, MID$, LEN, VAL, and ASC let you control and print only the part of the string you want.

For example, the short program below displays WAY.

```
10 N$ = "WAYNE"
20 PRINT LEFT$ (N$,3)
```

Such control over strings can be valuable, as you will soon learn. For more information on these commands, see Chapter 6.

MAXFILES

Sooner or later, you may need to have the program access more than one text file at a time. To do this, use the command MAXFILES. Its format is

[1]Copyright 1978, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

Figure 8-1 Max Files (Private Eye).

MAXFILES n

where n is the number of files you want OPEN (in use) at the same time. You can then OPEN a particular file with

```
10 PRINT CHR$(4); "OPEN"; "FILE NAME"
```

DOS allows up to 16 OPEN files. You will probably never use that many, but you should be aware of some important facts concerning OPEN text files.

When you boot DOS, it automatically sets aside enough memory space for three files to be active concurrently (MAXFILES 3). This is the default condition.

To have the computer set aside enough space for, say, six files, type

```
MAXFILES 6
```

This memory space is called a *file buffer*. The computer sets aside 595 bytes of memory for each file of the three default file buffers. (See the next section, "Disk Format," for definition and explanation of *byte*.)

A buffer is an intermediary between the computer and the disk data file (see Figure 8-2). When you retrieve information from the disk, 256 bytes at a time feed into the input part of the file buffer. Then the buffer supplies the information to you as needed.

The same thing happens in reverse order when you send information to the disk. The output part of the buffer accumulates 256 bytes of information, which then moves all at once to the disk.

Remember to CLOSE your files when you are finished using them. The CLOSE command forces whatever information remaining in the output buffer to be SAVEd to the disk. Failure to do this results in an incomplete file.

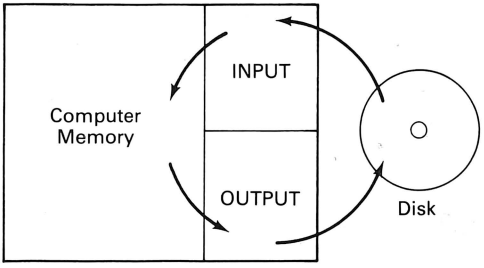


Figure 8-2 Computer Buffers.

Another important point: **Never** remove a disk containing an OPEN file from the drive. Always CLOSE the file first, as shown below.

```
300  PRINT CHR$(4); "CLOSE";"FILE NAME"
```

or

```
300  PRINT D$;"CLOSE";"FILE NAME"
```

If you don't, information left in the output buffer may be stored in the wrong place. We will discuss "CLOSE" in more detail later.

Another type of error, NO BUFFERS AVAILABLE, can occur if you try OPENing more active files than you established in the MAXFILES command. For example, suppose you had programmed MAXFILES 2, and two files were already active. An attempt to OPEN a third file would exceed the reserved memory and produce a NO BUFFERS AVAILABLE error message. Without error traps, this may stop the program. Any additional DOS commands (for example, CATALOG) would result in the same error message.

Since the MAXFILES command may interfere with a program, you should execute it before LOADing or RUNning the program. You can do this from a separate program.

For example, suppose you have a program named BILLING with five active text files. One way to set up the MAXFILES is as follows:

```
10  REM BILLING
20  REM SET UP MAXFILES
30  :
40  PRINT CHR$(4);"MAXFILES 5"
50  PRINT CHR$(4);"RUN BILLING.2"
```

This program sets up the correct number of MAXFILES and then automatically LOADs and RUNs your billing program, say, BILLING.2.

If you use MAXFILES in your main program instead, make sure it is the first statement in the program, before any string variables are declared, as in

```
10 PRINT CHR$(4);"MAXFILES 5"::
20 :
30 REM REST OF PROGRAM
```

DISK FORMAT AND DISK COMMANDS

Two more points we should cover before we discuss sequential or random-access text files are disk format and disk commands.

Disk Format

A little understanding of how information is stored on the floppy disk is helpful. The floppy is similar to a cassette tape, in that both are coated with magnetic materials and both store information. The advantage of the floppy over the cassette is speed (time required to load or save a file or program) and storage capacity.

The Apple disk drive stores almost 127,000 bytes of information on a DOS 3.3 16-sector diskette. A 13-sector diskette holds a little less. Sounds impressive, doesn't it? But what's a *byte?*

A byte is a unit of disk storage or computer memory. Each letter, number, control character, or symbol takes up one byte of storage space.

For example, in

```
10 A$ = "COMPUTER"
```

the word COMPUTER has 8 characters. It is an 8-byte word. The amount of storage space needed to store it on a disk would be 8 bytes + 1 byte for "overhead," or 9 bytes total. The extra byte is very important. It is the RETURN character, which is ASCII value 13. We will discuss that in more detail later.

Numbers work in the same way. For example, 15.32 equals 6 bytes of storage space, i.e., 4 numbers + 1 period + 1 overhead = 6 bytes.

The disk itself, inside a black paper cover, is 5 in. in diameter. (See Figure 8-3.)

It looks like a small record album in a jacket. Each diskette is divided into 35 or 40 concentric circles called *tracks.*

Each track is divided into 16 sectors. (See Figure 8-4.)

Each sector will store 256 bytes. As we save our data to the disk, it fills up each sector. The DOS (disk operating system) uses three tracks of the 35 or 40 available. It keeps track (pun intended) of what's where on the disk.

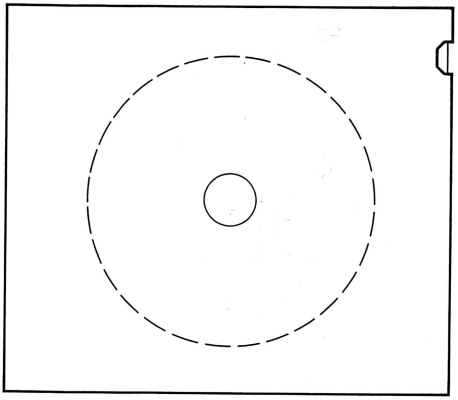


Figure 8-3 Floppy Disk.

Difference between Sequential and Random-Access Files

A sequential file differs from a random-access file.

The sequential file stores pieces of information one right after the other. We can demonstrate this with a picture. Imagine taking one track of a disk and laying it out straight.

Suppose you want to save three names,

JOHN
SALLY
SUE

They would be stored sequentially as shown in Figure 8-5 (* = RETURN character).

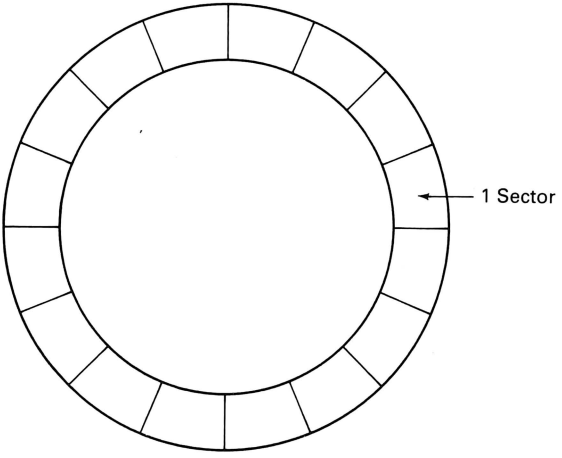


Figure 8-4 Track Arrangements.

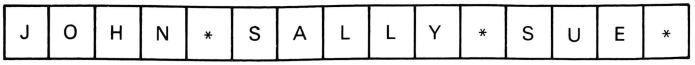


Figure 8-5 Sequential Text File Format.

The sequential file is very space-efficient. We normally use sequential files for permanent records such as telephone lists or other lists which require *little* or *no* change.

There is, however, a disadvantage to this type of file. If we want to find a particular name, for example, SUE, the disk drive must search through all the names that precede Sue. In other words, it begins with the first name, JOHN, then moves to SALLY, and then finally to SUE. If you have a long list of names, this could take quite a while.

A random-access file, on the other hand, has the opposite advantages and disadvantages, as we shall see.

For a random-access file, we must set the maximum length of each entry. The following example sets limits on two INPUT statements.

```
10 INPUT "ENTER YOUR NAME (20 CHARACTERS MAX.): ";N$
20 INPUT "ENTER YOUR AGE (3 CHARACTERS MAX.): ";AG$
```

Each combination of name and age is a ***record.*** Each record uses 25 bytes; that is,

Name: 20 bytes + 1 overhead = 21 bytes

Age: 3 bytes + 1 overhead = 4 bytes

Total = 25

In a random-access text file, you must set the record length before entering any data. This assigns all records the same number of bytes. We call it a *fixed length.* Thus all records in our example have a fixed length of 25 bytes whether or not we enter a character for each byte.

To see how this works, let's lay out one track of the disk again. We will use a fixed length of 7 for our example (* = RETURN character). See Fig. 8-6.

Note that the computer has set aside seven spaces for each record. Obviously, random-access files don't use disk space efficiently. The advantage over sequential

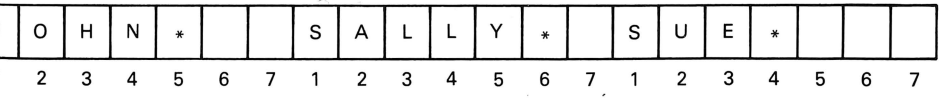


Figure 8-6 Random-access Text File Format.

files is access time. The computer locates a file directly without searching through each record ahead of it.

Disk Commands

You must give the computer special DOS commands to use the disk drive.

For example, to see the file names on a disk, type CATALOG and press the RETURN key. We issued this command in *immediate-execution mode*. This means (not surprisingly) that the computer executes the command immediately. Other commands we commonly use this way are LIST, SAVE, LOAD, RUN, and DELETE. (See Appendix C for a complete list of DOS commands.)

We can also put DOS commands in programs. For example, to make our HELLO program CATALOG the disk, type

```
10 REM HELLO PROGRAM
20 D$ = CHR$(4)
30 REM SLAVE DISK CREATED 4-2-80
40 PRINT D$;"CATALOG"
50 END
```

When we RUN this program, it automatically CATALOGs the disk just as though we had typed the command directly from the keyboard.

Issuing DOS commands from within a program is called *deferred-execution mode*. The command is deferred until the program reaches a specific line.

You can use many DOS commands, such as CATALOG, in either immediate-execution or deferred-execution mode. However, some DOS commands can be used only in the deferred-execution mode. These include OPEN, READ, WRITE, APPEND, and POSITION. Trying to type them in immediate-execution mode produces a NOT DIRECT COMMAND error message.

Note the PRINT D$ just ahead of the DOS command in line 40 of the CATALOG in the HELLO program.

D$ represents CTRL-D, which must precede every DOS command in a program. Line 20 makes ASCII value 4 (CTRL-D) equal to D$. This is certainly easier to write than a longer statement such as

```
40 PRINT CHR$(4);"CATALOG"
```

In older programs, CTRL-D is sometimes represented as

```
20 D$ = "": REM CTRL-D
```

That's because CHR$ was not available in INTEGER BASIC.

There is actually a CTRL-D between the quotation marks. You can't see it

because the computer does not display CONTROL characters. In fact, the cursor doesn't move forward either.

To enter the invisible CTRL-D in line 20, type:

```
20 D$ = "
```

then hold down the CTRL key while typing D. Now type the closing quotation mark and the rest of the line.

The semicolon in line 40 is optional, although you should include it for clarity. The quotation marks around CATALOG **are not optional.** Every DOS command must be enclosed in quotation marks. Some examples are

```
PRINT D$;"CATALOG"
PRINT D$;"OPEN FILE NAME"
PRINT D$;"SAVE FILE NAME"
etc.
```

etc.

Use only one DOS command per PRINT statement.

A final requirement for using a DOS command in a program is that a RETURN or carriage return (ASCII value 13, not the command) must precede the D$. For example,

```
 60 PRINT "DO YOU WANT TO SAVE THIS (Y/N)?";
 70 GET Y$
 80 IF Y$ = "N" THEN 10
 90 PRINT D$;"SAVE FILE NAME"
100:
110 REM REST OF PROGRAM
```

displays

```
SAVE FILE NAME
```

rather than executing the command. You must add

```
85 PRINT
```

or

```
70 GET Y$:PRINT Y$
```

to insert a RETURN before D$ in line 90.

Alternatives are PRINT CHR$(13) D$;"SAVE...." or PRINT CHR$(13) CHR$(4) ;"SAVE...".

Some programmers assign a variable to the two character strings, as we did with CTRL-D. This looks like

```
20 T$ = CHR$(13) + CHR$(4)
30 D$ = CHR$(4)
   :
   :
   :
70 GET Y$
80 PRINT T$;"SAVE ...."
```

The following program gives an example of another problem to watch out for.

```
10 REM TEST CATALOG PROGRAM
20 D$ = "": REM CTRL-D
30 PRINT "TEST";
40 PRINT D$;"CATALOG"
50 END
```

RUNning this program displays TEST CATALOG, because the semicolon at the end of line 30 suppresses the RETURN. Just delete the semicolon to make the CATALOG command execute properly.

Disk Command Used in Text Files

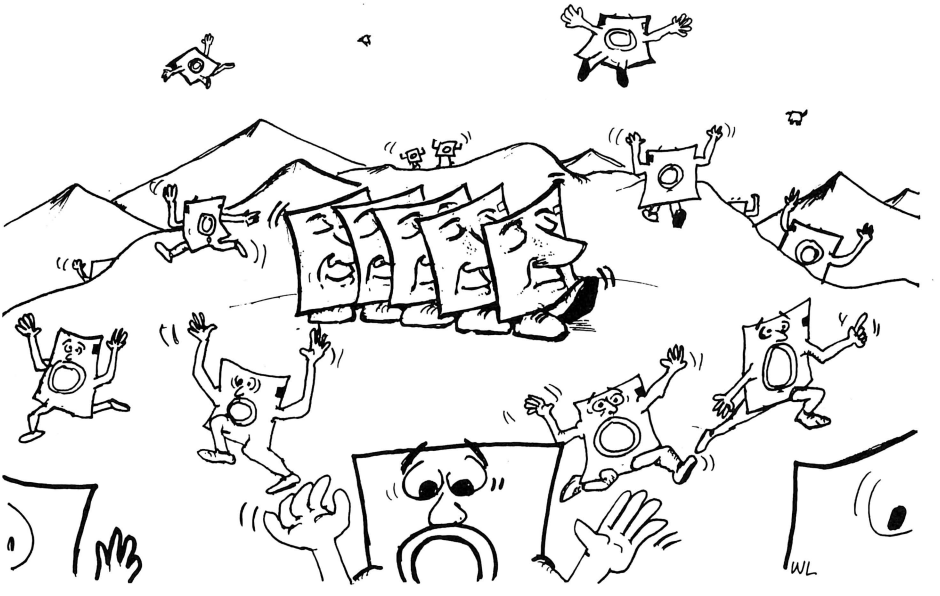
In a manual filing system, to look at a particular file, you must take its folder from a file cabinet drawer, then

1. **OPEN** the file folder.
2. **READ** its contents.
3. When you're finished, **CLOSE** the folder.

The commands to handle a text file on disk are similar. Below are the commands and a short explanation of each.

OPEN (Filename)	Opens the named text file
READ (Filename) and **INPUT**	Inputs (reads) data from the named file into the computer
WRITE (Filename) and **PRINT**	Prints your entered data into the named file
DELETE (Filename)	Deletes the named file from the disk
CLOSE	Closes the file

Both random-access and sequential text files use these commands.



9 Sequential Text Files

INTRODUCTION

Because they don't waste space, sequential text files are an excellent way to store data on disk. As we discussed in the last chapter, sequential text files are best for storing information that requires little or no change. Some examples are telephone lists, product inventory information, customer information, name and address lists, etc.

In this chapter, we will discuss how to

Create a sequential text file.
Write information to it.
Retrieve information from it.
Add information to the file.
Change stored data.
Use a few advanced techniques to manipulate files.

REM statements will sometimes refer to material covered elsewhere in the book. For example, in

```
20 INPUT "ENTER YOUR NAME: ";N$
30 REM ERROR TRAPS GO HERE (see Chapter 7)
```

line 30 reminds you to error trap for information entered in line 20. If you are unfamiliar with the procedure, refer to the chapter indicated in parentheses in the REM line.

We assume that you will always include an introduction module stating the program name, author, data, list of variables, etc. See Chapter 8 for details.

HOW TO CREATE A SEQUENTIAL TEXT FILE

We can summarize creation of sequential text files in four steps.

OPEN
WRITE
PRINT
CLOSE

To demonstrate how this works, let's create a program for a telephone list. We will develop it in four parts.

The program begins as follows:

```
 10 REM TELEPHONE LIST
 20 :
 30 REM REST OF INTRODUCTION MODULE (see Chapter 8)
 40 :
    :
    :
    :
150 D$ = CHR$ (4): REM CTRL-D
```

Step 1—Open

The first step is to **OPEN** a file. We will call ours TEL LIST. Type

```
200 PRINT D$; "OPEN TEL LIST"
```

OPEN assigns a buffer to TEL LIST. If the file doesn't exist, OPEN creates it.

Steps 2 and 3—Write and Print

The second and third steps involve **WRITE**ing the telephone list to the disk. "But," you say, "Wait a minute! What's the computer going to WRITE to the file? I haven't entered any telephone numbers yet."

Quite right! Before we can WRITE to the file, we must enter some information.

We want to enter several names and their telephone numbers. To make the program allow this, type

```
240 INPUT "ENTER NAME: ";N$
250 REM ERROR TRAP GOES HERE (see Chapter 7)
260 INPUT "ENTER TELEPHONE NUMBER: ";TN$
270 REM ERROR TRAP GOES HERE
```

Now that we can enter the first name and telephone number, let's WRITE them to the file.

```
290 REM WRITE TO FILE
300 PRINT D$; "WRITE TEL LIST"
310 PRINT N$: PRINT TN$
```

Note that D$ must precede the DOS command WRITE in line 300.

Any PRINT commands following WRITE (as in line 310) always PRINT the data to the text file, **not to the screen.** In other words, the WRITE command is like telling the computer:

> "PAY ATTENTION, COMPUTER! I want you to do something different with this data. I want it PRINTed to the text file on the disk, not the screen."

After the name and telephone number have been written to the text file, we need to again tell the computer,

> "HEY COMPUTER! PAY ATTENTION AGAIN! I want you to stop writing to the text file every time you see a PRINT statement. Now I want you to write any PRINT statement you see back to the screen, as you used to do."

The command that instructs the computer to do this is simply **PRINT D$.** No commas. No semicolons. For example:

```
330 REM TERMINATE WRITE OPERATION
340 PRINT D$
```

Our program at this point looks like

```
 10 REM TELEPHONE LIST
 20 :
 30 REM REST OF INTRODUCTION MODULE (see Chapter 8)
 40 :
    :
    :
    :
150 D$ = CHR$(4): REM CTRL-D
    :
    :
200 PRINT D$; "OPEN TEL LIST"
    :
    :
240 INPUT "ENTER NAME: ";N$
250 REM ERROR TRAP GOES HERE (see Chapter 7)
260 INPUT "ENTER TELEPHONE NUMBER: ";TN$
270 REM ERROR TRAP GOES HERE
    :
290 REM WRITE TO FILE
300 PRINT D$; "WRITE TEL LIST"
310 PRINT N$: PRINT TN$
320 :
330 REM TERMINATE WRITE OPERATION
340 PRINT D$
```

Now we can put one name and telephone number in our file. But that isn't much of a list. We need to WRITE program lines to allow additional entries.

We can do this with a loop, that is, sending the program back to enter more data and WRITE it to the file. To let the program know when all the data has been entered, add

```
220 PRINT "TYPE 'END' INSTEAD OF NAME TO END": PRINT "DATA ENTRY."
245 IF N$ = "END" THEN 380
350 GOTO 220
```

Line 350 sends the program back to repeat the data entry process.

Line 220 asks for END to stop the data entry process.

Line 245 sets a condition: If END is entered, then go to line 380, which will begin the fourth and final step.

Step 4—Close

After entering all the data, it is time to **CLOSE** the file. You must **always** CLOSE a file when you're through with it (see MAXFILES, Chapter 8, for more information). We add these lines to CLOSE:

```
360 :
370 REM CLOSE FILE
375 :
380 PRINT D$; "CLOSE TEL LIST"
```

Again, since CLOSE is a DOS command within a program, it must be preceded by PRINT D$;.

As your programs get larger, you may have several text files OPEN at one time. Each may be CLOSEd individually without affecting any other open file, as in

```
PRINT D$; "CLOSE TEL LIST"
```

or

```
PRINT D$; "CLOSE ";F$
```

where F$ is a string that represents a file name.

You may also CLOSE *all* the files at the same time with

```
PRINT D$; "CLOSE"
```

We could have used this command in line 380 of our telephone list program, since it was the only file we had OPEN at the time. Be careful, however. Do not use CLOSE by itself unless you are sure you want to CLOSE every open file.

Our completed program looks like this:

```
 10 REM TELEPHONE LIST
 20 :
 30 REM REST OF INTRODUCTION MODULE (see Chapter 8)
 40 :
    :
    :
    :
150 D$ = CHR$(4): REM CTRL-D
    :
    :
200 PRINT D$; "OPEN TEL LIST"
    :
220 PRINT "TYPE 'END' INSTEAD OF NAME TO END": (PRINT "DATA ENTRY"
    :
240 INPUT "ENTER NAME: ";N$
245 IF N$ = "END" THEN 380
250 REM ERROR TRAP GOES HERE (see Chapter 7)
260 INPUT "ENTER TELEPHONE NUMBER: ";TN$
```

```
270 REM ERROR TRAP GOES HERE
    :
290 REM WRITE TO FILE
300 PRINT D$; "WRITE TEL LIST"
310 PRINT N$: PRINT TN$
320 :
330 REM TERMINATE WRITE OPERATION
340 PRINT D$
350 GOTO 220
360 :
370 REM CLOSE FILE
375 :
380 PRINT D$; "CLOSE TEL LIST"
390 END
```

While the program is RUNning, the computer does not display the data being written to the text file. To see it, type MON C, I, O and press RETURN before running the program. This puts the computer in MONitor mode (see Chapter 6 for more information on MON and NOMON).

The program in MON would appear as follows:

```
OPEN TEL LIST
WRITE TEL LIST
(name)
(telephone number)
CLOSE TEL LIST
```

Since MON isn't usually on, the program should let the user know what is happening while the screen is blank. You can do this with a line such as

```
280 HOME: PRINT "PLEASE WAIT WHILE DATA IS SAVED."
```

Try RUNning the program. Enter 10 names and telephone numbers.

You have now successfully created and written to a sequential text file. In our sample program, we wrote several names and telephone numbers to TEL LIST. Note that each PRINT statement without an ending comma or semicolon has an automatic RETURN.

For example, if

```
N$ = JOHN and TN$ = 555-2003
```

then

```
310 PRINT N$: PRINT TN$
```

would be stored on the disk as shown in Figure 9-1 (* = Return character).

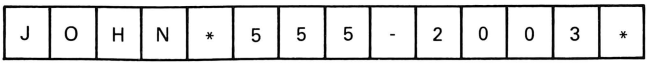


Figure 9-1 Sequential Format.

Each item of data ending with a RETURN is called a **field.** JOHN* is one field and 555-2003* is another. The two data items together are called a **data set.**

Our data sets may vary in size, depending on how many fields they contain. The fields, also, may vary in length depending on the length of the name or telephone number.

? EXTRA IGNORED

Using commas or colons in strings can create problems, as in the following:

```
240 INPUT "ENTER NAME: ";N$
```

Responding to the prompt with

```
ENTER NAME: RHODERICK, JOHN
```

produces the error message ?EXTRA IGNORED. The computer only accepts RHODERICK, ignoring the comma and JOHN.

One way to overcome this problem is to put the name in quotation marks; that is,

```
ENTER NAME: "RHODERICK, JOHN"
```

The computer will ignore the quotation marks but SAVE the entire name, including the comma. However, using quotation marks here is inconvenient.

Another solution is to write a subroutine that accepts commas or colons. Chapter 5 lists a few.

RETRIEVING INFORMATION FROM A SEQUENTIAL TEXT FILE

Our sample sequential text file contains a telephone list. Let's now write a program to retrieve data from it.

The steps we use are

OPEN
READ

INPUT
CLOSE

Before OPENing a file and READing it, you must name the type of variable to be entered (i.e., string or numeric). INPUT will not accept a string if it is expecting a numeric. Remember to define the variables in the introduction module so that you can keep track of them.

Step 1—Open

This step is the same as the one we used in writing to the file.

```
 10 REM RETRIEVE TELEPHONE LIST
 20 :
 30 :
 40 :
    :
    :
100 D$ = CHR$(4): REM CTRL-D
110 PRINT D$; "OPEN TEL LIST"
```

Steps 2 and 3—Read and Input

The second and third steps retrieve the information from the disk. Keeping in mind the specified variable type (string or numeric), add the following lines.

```
130 PRINT D$; "READ TEL LIST"
140 INPUT N$, TN$
150 REM TERMINATE READ OPERATION
160 PRINT D$
```

Line 130 presents the DOS command **READ.** PRINT D$ must precede "READ TEL LIST".

READ is similar to the WRITE statement.

Issuing a READ command makes **any** INPUT or GET statement that follows (as in line 140) INPUT data from the text file on the disk, *not* from the keyboard.

To make the computer stop INPUTing data from the disk, type PRINT D$ (as in line 160). The computer will now resume looking for INPUT data from the keyboard.

So far, our program looks like this:

```
 10 REM RETRIEVE TELEPHONE LIST
 20 :
 30 :
```

```
 40 :
    :
    :
100 D$ = CHR$(4): REM CTRL-D
110 PRINT D$; "OPEN TEL LIST"
120 :
130 PRINT D$; "READ TEL LIST"
140 INPUT N$, TN$
150 REM TERMINATE READ OPERATION
160 PRINT D$
```

In the above program, the computer READs the program's first data set into its memory. As with creating a sequential text file, nothing appears on the screen. We must give the program instructions for displaying the data set.

The format is up to you. We will display our telephone list as follows:

```
PAM MAGNUSSON     555-2003
BRUCE ANDERSON    555-5931
MARK LABARGE      555-1241

 etc.
```

To begin with the process, type

```
180 PRINT N$; TAB(20) TN$
```

This displays the first data set. But our program is not yet complete. We still need to READ and display the remaining data sets.

To do this, we need a loop, just as we did when entering the data sets. Add

```
200 GOTO 130
```

to send the program back to INPUT the remaining data sets.

After all the data sets have been INPUT, line 200 loops back (as usual) to line 130. This final loop stops the program and produces an END OF DATA error message, because it is trying to read in more data than is in the file.

We can prevent this by using an ONERR statement. (If you are unfamiliar with ONERR, read Chapter 6 before proceeding.)

```
120 ONERR GOTO 220
130 PRINT D$; "READ TEL LIST"
    :
    : REM REST OF PROGRAM
    :
    :
```

```
220 LET Y = PEEK (222)
230 IF Y < > 5 THEN GOTO 270
240 PRINT D$; "CLOSE TEL LIST"
250 PRINT: PRINT "ALL ENTRIES LISTED."
260 END
270 PRINT "AN ERROR HAS OCCURRED IN LINE "; PEEK (218) +
    PEEK (219) * 256
280 PRINT "ERROR CODE: ";Y
290 PRINT D$; "CLOSE"
300 END
```

Line 120 sets an error trap. From there on, an error returns the program to line 220.

Lines 220 and those following handle the error. The END OF DATA error code is 5.

Line 220 sets the variable Y to PEEK (222), the memory location of error codes.

Lines 230–260 handle the END OF DATA error.

Lines 270–300 handle error codes other than 5 (END OF DATA). Line 270 prints the line number in which the error occurred. Line 280 prints the error code itself.

ADDING INFORMATION TO A SEQUENTIAL TEXT FILE

Often you will need to add information to a text file SAVEd on disk. For example, telephone, inventory, and customer lists must be updated continually.

To add information to a sequential text file, we will use a new DOS command called **APPEND.**

OPEN will not work here, because it sets an internal pointer to byte 0, the first character in the file. The pointer advances one byte with each character entered, overwriting any data already in the file.

You can picture the pointer, in your mind, as a little arrow that says to the computer, "WRITE the data right here where I point." For example (* = return character), see Figure 9-2.

APPEND, on the other hand, opens an existing file and moves the pointer one byte beyond the last character in the file. See Figure 9-3.

The format to add data is as follows:

```
APPEND
WRITE
```

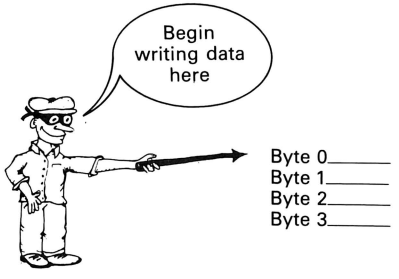


Figure 9-2 (OPEN example)

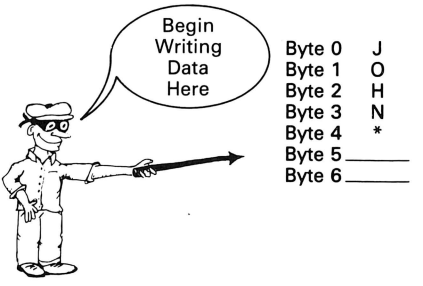


Figure 9-3 (APPEND example)

```
PRINT
CLOSE
```

Never use a READ command after APPEND.

The following program demonstrates how to add names and telephone numbers to TEL LIST.

```
 10 REM TELEPHONE LIST
    :
    :
    :
150 D$ = CHR$(4): REM CTRL-D
    :
    :
    :
400 REM ADD INFORMATION TO FILE
410 INPUT "ENTER NAME: ";N$
420 INPUT "ENTER TELEPHONE NUMBER: ";TN$
    :
    :
    :
500 REM APPEND FILE
510 PRINT D$; "APPEND TEL LIST"
520 PRINT D$; "WRITE TEL LIST"
530 PRINT N$: PRINT TN$
540 PRINT D$; "CLOSE"
```

Of course, you can only use APPEND to open an existing file. Trying to create a file with APPEND results in a FILE NOT FOUND error.

While it is true that you cannot create a file with APPEND, you can streamline a program by using it to enter the first data items. Let's see how this works with TEL LIST.

```
 10 REM TELEPHONE LIST
    :
    :
    :
150 D$ = CHR$(4): REM CTRL-D
    :
    :
200 PRINT D$; "OPEN TEL LIST"
210 PRINT D$; "CLOSE TEL LIST"
220 :
230 REM PUT DATA ENTRY PROGRAM LINES HERE
    :
    :
    :
290 :
300 PRINT D$; "APPEND TEL LIST"
310 PRINT D$; "WRITE TEL LIST"
320 PRINT N$: PRINT TN$
330 PRINT D$; "CLOSE"
340 PRINT: PRINT "DATA SUCCESSFULLY APPENDED"
350 END
```

Lines 200–210 are the key to this program. Line 200 creates the file TEL LIST. Line 210 CLOSEs it. Even though it contains no data, the file now exists. This is the only requirement for using APPEND.

Lines 230–290 let you enter names and telephone numbers (as in earlier examples).

Line 300 uses APPEND to open TEL LIST and move the pointer one byte beyond the last character entered (same as the beginning, in this case, since the file is empty). As a result, the pointer places itself at byte 0. Thus we don't need OPEN to begin entering data.

HOW TO MAKE CHANGES IN A SEQUENTIAL TEXT FILE

Because there are no unused spaces in sequential text files, we cannot change these files easily. Suppose, for example, you learn that SAM JONES prefers to be called SAMUEL. Changing a four-byte word, SAM* (* = RETURN character), as shown

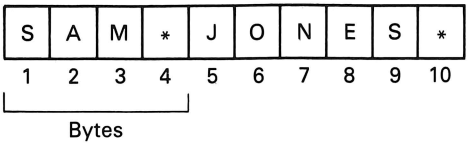


Figure 9-4 Sequential Byte Explanation.

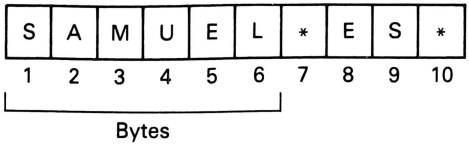


Figure 9-5 Sequential Byte Overwriting.

in Fig. 9-4, to a seven-byte word, SAMUEL*, as shown in Fig. 9-5, overwrites part of the entry that follows. That is, U, E, L, and * (RETURN) write over the * (RETURN), J, O, and N in JONES. You can easily see the havoc that this can create in a file.

To overcome this problem, we must create a temporary text file, as follows:

1. OPEN TEL LIST
2. OPEN Temporary file
   Let's call it "TEMP."
3. INPUT name to be changed.
   In this case, SAM.
4. Search for old name.
5. WRITE to the TEMP file any data set (name and telephone number) that doesn't match the old name in Step 3 and is to remain unchanged.
6. If the old name is not found, display a message indicating so. CLOSE the file and return to Step 1.
7. Display the name.
8. Make the necessary changes.
9. WRITE the new information to the TEMP file.
10. WRITE the remaining data sets in TEL LIST that are to remain unchanged to the TEMP file.
11. CLOSE both files.
12. DELETE TEL LIST.
13. RENAME TEMP to TEL LIST.

Our program could be written as follows:

```
 10  REM CHANGE INFORMATION IN TEL LIST
     :
     :
     :
150  D$ = CHR$ (4): REM CTRL-D
     :
     :
     :
```

```
200 REM MAKE CHANGES
205 HOME
210 PRINT "TYPE 'EXIT' TO STOP"
215 INPUT "ENTER OLD NAME TO CHANGE: ";NM$
220 IF NM$ = "EXIT" THEN GOTO 830
225 REM ERROR CHECKS
230 PRINT D$; "OPEN TEL LIST"
240 PRINT D$; "OPEN TEMP"
245 ONERR GOTO 325
250 PRINT D$; "READ TEL LIST"
260 INPUT N$: INPUT TN$
270 PRINT D$
280 IF NM$ = N$ THEN 410
290 PRINT D$; "WRITE TEMP"
300 PRINT N$: PRINT TN$
310 PRINT D$
320 GOTO 250
325 REM ONERR ROUTINE
330 PRINT "UNABLE TO LOCATE ";NM$: PRINT "PLEASE CHECK SPELLING."
340 PRINT D$; "CLOSE"
350 PRINT D$; "DELETE TEMP"
360 GOTO 210
    :
    :
400 REM NAME FOUND
410 HOME: PRINT "NAME FOUND!": PRINT:PRINT
420 PRINT NM$;TAB (20) TN$
430 PRINT:PRINT "PRESS <RETURN> IF NO CHANGE WANTED."
    : PRINT
440 PRINT "CHANGE ";NM$;: INPUT " TO: ";N$
450 IF N$ = "" THEN N$ = NM$
460 PRINT "CHANGE ";TN$;: INPUT " TO: ";TE$
470 IF TE$ = "" THEN TE$ = TN$
480 HOME: PRINT "PLEASE WAIT WHILE FILE IS BEING UPDATED."
    :
500 REM WRITE NEW INFO TO TEMP FILE
510 PRINT D$; "WRITE TEMP"
520 PRINT N$: PRINT TE$
530 PRINT D$
550 REM PRINT REMAINING DATASETS TO TEMP
560 ONERR GOTO 700
570 PRINT D$; "READ TEL LIST"
580 INPUT N$: INPUT TN$
590 PRINT D$
600 PRINT D$; "WRITE TEMP"
610 PRINT N$: PRINT TN$
620 PRINT D$
```

```
630 GOTO 570
    :
    :
    :
700 REM CLOSE FILES
710 IF PEEK (222) = 5 THEN 730
720 PRINT "ERROR ";PEEK (222);" ON LINE ";PEEK (218) + PEEK (219) * 256:
    PRINT D$;"CLOSE": END
730 PRINT D$;"CLOSE"
740 REM DELETE OLD FILE
750 PRINT D$; "DELETE TEL LIST"
760 REM RENAME TEMP
770 PRINT D$;"RENAME TEMP, TEL LIST"
780 HOME: PRINT "FILE UPDATED."
790 PRINT: PRINT
800 PRINT "DO YOU WANT TO MAKE ANY MORE CHANGES (Y/N)?:";
810 GET Y$: PRINT Y$
820 IF Y$ = "Y" THEN 205
830 END
```

One final comment: Constantly overwriting data (e.g., game scores) increases the possibility of the READ loop's including leftover data from the end of the file. To eliminate this possibility, add

```
100 PRINT D$; "OPEN (filename)"
110 PRINT D$; "DELETE (filename)"
120 PRINT D$; "OPEN (filename)"
130 :
140 REM REST OF PROGRAM
```

If the old file exists, these lines DELETE it.

DELETE TEXT FILES

The form for **DELETE**ing a sequential text file from immediate command mode is

DELETE (filename)

From deferred execution mode it is

```
20 PRINT D$; "DELETE "; F$
```

where F$ equals the filename.

COPYING TEXT FILES

You can copy and transfer text files to another disk by using the FID program on your system master diskette. The procedure is as follows:

1. BRUN FID.
2. When FID's menu appears, select OPTION 1, COPY FILES, and press <RETURN>.
3. You will be prompted for "Source slot". Respond by typing "6". Respond to the "Drive" prompt by typing "1'.
4. Answer "6" and "1", respectively, to the "destination slot" and "drive number" prompts (or "6" and "2" if computer has two disk drives).
5. The next prompt will ask for the filename. Type the name exactly as it appears in the CATALOG.

   Example

   FILENAME? TEL LIST

   Then press <RETURN>.

   If, at this point, you don't remember the exact name, then press any key and <RETURN>. The next prompt allows you to exit the program by pressing <ESC>. This recalls the menu. Now you can use OPTION 2 to CATALOG the disk before returning to OPTION 1.
6. After you have entered the correct filename, remove the diskette with FID on it from the drive. Note: If your computer has a second drive, insert the destination diskette.
7. Insert the source diskette in drive 1. Close the drive door and press <RETURN>.
8. The filename will appear at the top of the screen. Press <RETURN> again.
9. Drive 1 will search for the named file. When it finds it, the drive will stop. In response to the prompt, remove the source diskette and replace it with the destination diskette. Press <RETURN>.

   Note: If your computer has two disk drives, the copy program will function automatically to completion. Skip to step 11.
10. If the file is very long, you may be asked to switch back and forth between the source and destination diskette until the copying is complete.
11. When the copying is finished, DONE will appear on the screen. Press <RETURN> to recall the menu.
12. If you have other files to copy, repeat these steps.
13. When all copying is completed, press OPTION 9 to exit the FID program.

OTHER WAYS TO COPY FILES

You can also write a program to copy one file to another, as follows:

```
 10  REM COPY PROGRAM
 20  :
 30  :
     :
     :
100  D$ = CHR$ (4): REM CTRL-D
120  HOME
130  INPUT "ENTER SOURCE FILE NAME: ";F1$
140  INPUT "ENTER DESTINATION FILE NAME: ";F2$
145  PRINT D$; "OPEN ";F1$
150  PRINT D$; "OPEN ";F2$
160  PRINT D$; "DELETE ";F2$
170  PRINT D$; "OPEN ";F2$
180  ONERR GOTO 260
190  PRINT D$; "READ ";F1$
200  INPUT N$: INPUT TN$
210  PRINT D$
220  PRINT D$; "WRITE ";F2$
230  PRINT N$: PRINT TN$
240  PRINT D$
250  GOTO 190
260  IF PEEK (222) < > 5 THEN PRINT "ERROR "; PEEK (222):
     "ON LINE "; PEEK (218) + PEEK (219) * 256
270  PRINT D$; "CLOSE"
280  PRINT: PRINT "FILES CLOSED."
290  END
```

To copy individual data sets, add these lines:

```
212  HOME
214  PRINT N$; TAB (20) TN$
216  PRINT:PRINT "DO YOU WISH TO COPY THIS DATASET (Y/N)?: ";
217  GET Y$: IF Y$ = "N" THEN PRINT: GOTO 250
218  PRINT
```

With a few modifications, you can even use this program to edit. You could then go through each entry or data set one at a time, making changes as necessary. As we learned earlier, this involves rewriting to a temporary file, then deleting the original file and RENAMEing the temporary file to the original.

The program with modifications for editing looks like

```
 10  REM COPY PROGRAM
 20  :
```

```
 30 :
    :
 50 HOME: PRINT TAB (09) "SEQ. FILE EDITING PROGRAM"
 70 PRINT:PRINT
    :
100 D$ = CHR$ (4): REM CTRL-D
120 HOME
130 INPUT "ENTER SOURCE FILE NAME: ";F1$
132 ONERR GOTO 360
135 PRINT D$; "VERIFY ";F1$
140 F2$ = "TEM"
145 PRINT D$; "OPEN ";F1$
150 PRINT D$;"OPEN ";F2$
160 PRINT D$; "DELETE ";F2$
170 PRINT D$;"OPEN ";F2$
180 ONERR GOTO 370
190 PRINT D$;"READ ";F1$
200 INPUT N$: INPUT TN$
210 PRINT D$
212 HOME
214 PRINT N$; TAB (20) TN$
216 PRINT:PRINT "DO YOU WANT TO COPY THIS DATA SET (Y/N)?: ";
217 GET Y$: PRINT Y$: IF Y$ = "N" THEN GOTO 350
240 PRINT "DO YOU WANT TO MAKE ANY CHANGES?: ";
250 GET Y$: PRINT Y$: IF Y$ = "N" THEN NM$ = N$: TE$ = TN$:
    GOTO 320
260 PRINT: PRINT "PRESS <RETURN> FOR NO CHANGE.": PRINT
270 PRINT "CHANGE ";N$;" TO: ";:INPUT "";NM$
280 IF NM$ = "" THEN NM$ = N$
290 PRINT "CHANGE ";TN$;" TO: ";: INPUT "";TE$
300 If TE$ = "" THEN TE$ = TN$
310 PRINT
320 PRINT D$;"WRITE ";F2$
330 PRINT NM$: PRINT TE$
340 PRINT D$
350 GOTO 190
360 IF PEEK (222) = 6 THEN CALL -198: PRINT "FILE
    DOESN'T EXIST. REENTER.": PRINT: GOTO 130
370 IF PEEK (222) < > 5 THEN PRINT "ERROR ";PEEK (222);" ON
    LINE "; PEEK (218) + PEEK (219) * 256: PRINT D$;"CLOSE":
    END
380 PRINT D$; "CLOSE"
390 PRINT: PRINT "FILES CLOSED."
400 PRINT
410 PRINT D$;"DELETE ";F1$
420 PRINT D$;"RENAME ";F2$;",";F1$
430 PRINT "FILE ";F1$
440 PRINT "HAS BEEN CORRECTED AND UPDATED."
450 END
```

ALPHABETIZING TEXT FILES

The following procedure lets you put files in alphabetical or numerical order.

1. Create a temporary file.
2. Alphabetize the contents of the original file.
3. Write the alphabetized list to a temporary file.
4. DELETE the original source file.
5. RENAME the temporary file back to the name of the original.

There are many sorting programs. Some are faster than others. But most evaluate an expression (word or number) in terms of ASCII decimal values (see APPENDIX A).

For example, if you press Shift A for a capital A, the computer recognizes it as 65, the ASCII value of capital A. Since the computer translates each key pressed into its ASCII equivalent, we can alphabetize by ASCII numerical order.

For example, the word "FIRST" has the ASCII values shown in Figure 9-6.

ASCII values for the word "ADAM" are shown in Figure 9-7.

The computer alphabetizes FIRST and ADAM numerically by ASCII value. Since 65 (the ASCII equivalent of A) is less than 70 (the ASCII equivalent of F), ADAM sorts ahead of FIRST.

The computer sorts similar words, such as

```
SALES
SALE
```

by comparing characters left to right until it finds two that are different. Let's see how this works. See Figures 9-8 and 9-9.

The computer compares the first four letters of each word and determines that they are alike. It goes on to the fifth character, finding the final S in SALES and nothing in SALE. No character at all is called a *null character*. In ASCII, null always comes before any character, so SALE sorts ahead of SALES.

By looking at the ASCII equivalents listed in Appendix A, you can see that numbers have lower values than letters, punctuation marks have lower values than numbers, and so on. This is important to remember when setting up a sort routine. If you understand how the computer interprets characters, you aren't as likely to get unexpected results.

Letter	F	I	R	S	T
ASCII Code	70	73	82	83	84

Figure 9-6 ASCII Assignment.

Letter	A	D	A	M
ASCII Code	65	68	65	77

Figure 9-7 ASCII Assignment.

ASCII Code	83	65	76	69	
Letter	S	A	L	E	← Nothing in This Block
Letter	S	A	L	E	S ← Something in This Block
ASCII Code	83	65	76	69	83

Figure 9-8 ASCII Comparison.

Figure 9-9 ASCII Comparison.

SORT ROUTINE

A sort routine is best used as a subroutine. If, for example, a text file contains several names, you would do the following:

1. INPUT all names and assign each to an array.
2. Send the names to a sort subroutine.
3. PRINT all the names in alphabetical order.

A program might look like this:

```
 10 REM SORT PROGRAM
 20 D$ = CHR$ (4) : REM CTRL-D
 30 PRINT D$;"OPEN TEXT FILE"
 40 PRINT D$; "READ TEXT FILE"
 50 INPUT R1: REM R1 = NUMBER OF NAMES IN FILE
 60 DIM N$ (R1)
 70 FOR J = 1 TO R1
 80 INPUT N$ (J)
 90 NEXT J
100 PRINT D$
110 PRINT
```

```
120  GOSUB 500: REM GO TO SORT SUB
       :
       :
150  REM REST OF PROGRAM
       :
       :
       :
400  END

500  REM SORT ROUTINE SUB
510
520
```

For examples of sort routines, see Chapter 7.

You may want to store the alphabetical list in another text file.

POSITION

Gaining access to information in a sequential text file, particularly if it is near the end of a file, can be a slow process. There is a way, however, to speed it up if you know exactly where the data is stored.

POSITION is a DOS command that lets you ". . . access information from any specified field within a text file."[1] A field, as you remember, is any item of data ending with a RETURN character (ASCII code 13).

The command's form is

POSITION (file name), Rp

where Rp is the relative field position.

Note: The R parameter we use here is different from the R for "Record" used in random-access text files (see next chapter).

An example in a program might be

```
20 PRINT D$;"POSITION TEL LIST,R2"
```

Since POSITION is a DOS command, we must precede it by a **PRINT D$;** statement.

Position affects the internal pointer in the same way the commands OPEN and APPEND do. OPEN moves the pointer to the first byte in the file. APPEND moves it to one byte beyond the last character.

[1]Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

Whereas OPEN and APPEND move the pointer to a specific place in the file, POSITION moves the pointer foward to the *pth field* ahead of the *current* pointer position. "If P = 0, the following READ or WRITE begins in the current field. If P = 1, the following READ or WRITE skips the current field and begins in the *next* field. If P = 2, the following READ or WRITE skips two fields including the current field, before beginning to READ or WRITE. And so on."[2]

Since POSITION is relative to the current pointer position, it is important to know exactly where the pointer is, or should be, before the command is issued. If the pointer is in the middle of the file and you issue a "POSITION (filename), R3" and expect the third field from the beginning of the file to be displayed, you would be greatly disappointed.

Unfortunately, POSITION moves the pointer only forward, not backward. But its plus is that it allows you to READ or WRITE beginning in any field of the file. Should you, however, try to POSITION the pointer beyond the end of the file, and END OF DATA error will occur.

POSITION must precede a READ or WRITE command. You can use it only in a file that has already been OPENed.

Let's use our TEL LIST files to demonstrate how POSITION works. POSITION will scan the file byte-by-byte looking for the Rpth RETURN character.

Suppose the contents of our sequential text file looks like that in Figure 9-10 (* = RETURN character).

We can write a program using POSITION as follows:

```
 10 REM TEL LIST
 20 D$ = CHR$(4): REM CTRL-D
 30 PRINT D$; "OPEN TEL LIST"
 40 PRINT D$; "POSITION TEL LIST,R1"
 50 PRINT D$; "READ TEL LIST"
 60 INPUT TE$
 65 PRINT TE$
 70 PRINT D$;"POSITION TEL LIST, R5"
 80 PRINT D$;"READ TEL LIST"
 90 INPUT TN$
 95 PRINT TN$
100 PRINT D$;"CLOSE"
```

Refer to Figure 9-10 as we discuss the program. What will lines 65 and 95 display?

Line 30 OPENs the file TEL LIST and sets the pointer to the first byte in the file (the beginning of field 1), specifically the letter J in JOHN.

[2]Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

Line 40 looks for the first RETURN character (,R1) and advances the pointer to the beginning of the next field.

The first RETURN character it finds is at the end of the field that contains JOHN, so the pointer moves to the beginning of field 2.

Line 50 begins the READing process.

Line 60 INPUTs the telephone number (field 2) and sets it to the variable TE$. The pointer advances to the beginning of field 3.

Line 65 displays TE$, that is, 555-2002.

```
Field 1  JOHN*
Field 2  555-2002*
Field 3  SUE*
Field 4  555-3506*
Field 5  FRANK*
Field 6  555-8495*
Field 7  KIM*
Field 8  555-7220*
Field 9  TERRI*
Field 10 555-1116*
Field 11 SYBIL*
Field 12 555-9482*
```

Figure 9-10 Contents of File.

If we had MON C, I, O up to this point, it would look like this:

```
OPEN TEL LIST
POSITION TEL LIST,R1
READ TEL LIST
? 255-2002
255-2002
```

Line 70 looks for the fifth RETURN character and advances to the field following it.

Before you read on, which field does the pointer move to?

If you answered field 8, you are correct. Remember, after INPUTing TE$, the pointer finds itself at the beginning of field 3. It moves foward past the five RETURN characters of fields 3, 4, 5, 6, and 7. It is then at the beginning of field 8, telephone number 555-7220.

Line 80 begins the READ process.

Line 90 INPUTs the field 555-7220 and sets it to variable TN$.

Line 95 displays TN$.

Line 100 CLOSEs the file.

The program displays the two telephone numbers represented by TE$ AND TN$.

POSITION is an excellent way to format data in any order, if you know the contents of the file. For example, the following program displays only the names in TEL LIST.

```
 10 REM TEL LIST
 20 D$ = CHR$(4): REM CTRL-D
 30 PRINT D$;"OPEN TEL LIST"
 40 INPUT D$;"READ TEL LIST"
 60 INPUT N$
 70 PRINT N$
 75 ONERR GOTO 210
 80 PRINT D$;"POSITION TEL LIST,R1"
 90 PRINT D$;"READ TEL LIST"
100 INPUT N$
110 PRINT N$
120 GOTO 80
    :
    :
200 REM ERROR TRAP
210 IF PEEK (222) <> 5 THEN CALL -198: PRINT "UNUSUAL ERROR: ";
    PEEK (222)
220 PRINT D$;"CLOSE"
230 END
```

Line 30 OPENs the file and puts the pointer at the first byte.

Line 60 INPUTs JOHN.

Line 80 skips the next field (field 2).

Line 100 INPUTs SUE (field 3).

Lines 70 and 110 display the names.

Line 120 creates a loop which repeats the skip process.

Line 75 creates an ONERR trap.

Line 210 displays any errors other than code 5-END OF DATA.

Line 220 CLOSEs the file.

Now, let's control which name appears first. Suppose we want KIM (field 7) to appear first and SUE (field 3) to appear second.

Since POSITION cannot move the pointer backward through the file, how can we handle this? It's easy. The following program shows how.

```
 10 REM TEL LIST
 20 D$ = CHR$ (4)
 30 PRINT D$;"OPEN TEL LIST"
 40 PRINT D$;"POSITION TEL LIST,R6"
 50 PRINT D$;"READ TEL LIST"
 60 INPUT N$
 65 PRINT N$
 70 PRINT D$;"OPEN TEL LIST"
 80 PRINT D$;"POSITION TEL LIST,R2"
 90 PRINT D$;"READ TEL LIST"
100 INPUT N$
110 PRINT N$
120 PRINT D$;"CLOSE"
```

Line 30 OPENs the file and moves the pointer to the beginning of the file (byte 0).

Line 40 advances the pointer to field 7. Lines 60 and 65 INPUT and PRINT KIM's name to the screen.

Line 70 OPENs the file again, which puts the pointer back at the beginning of the file.

Line 80 advances the pointer to field 3.

Line 100 INPUTs SUE's name.

Line 110 PRINTs SUE's name.

BYTE CONTROL

It is possible to gain even greater control over a sequential text file by specifying a particular byte to WRITE or READ. As discussed earlier, a byte contains the ASCII equivalent of any character. That means it could be a punctuation mark, a symbol, a letter, a number, a RETURN character, space, and the like.

The form is

READ (filename), B(a number)

In a program it might look like

```
20 PRINT D$;"READ TEL LIST,B5"
```

The B parameter is an actual or absolute position in the file unless R is specified. If R is given, the B parameter is the actual position within the specified *field*.[3]

[3]Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

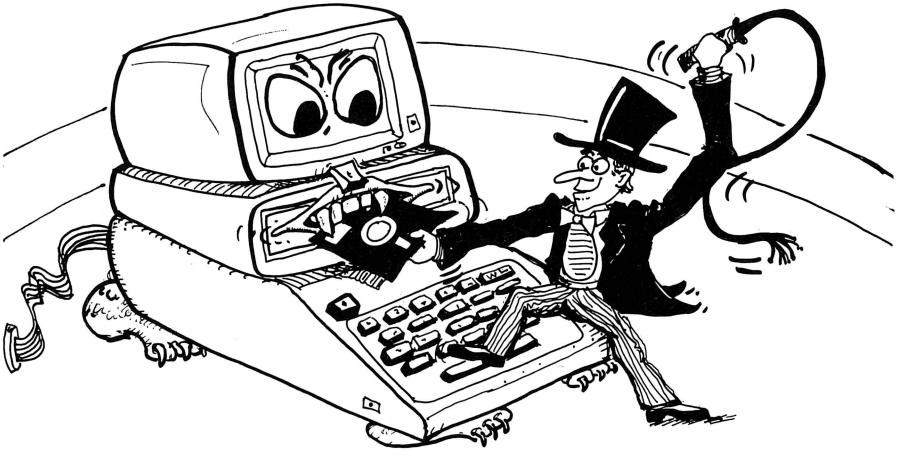


Figure 9-11 Byte Control.

In line 20 the pointer moves to the sixth byte of the file (remember, the first byte is 0).

Let's use the names from Figure 9-10 to explain the B parameter. The first two fields are

* = Return character

	Field 1					Field 2								
	J	O	H	N	*	2	5	5	-	2	0	0	2	*
Character Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Actual Byte Name	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 9-12 Byte and Field.

When we type

```
20 PRINT D$;"READ TEL LIST,B1"
30 INPUT N$
```

we are instructing the computer to begin READing from the actual byte name 1. This is character position 2, or the letter O.

To avoid confusion, always keep in mind that you should add 1 to the byte number to determine character position.

For example, if we read the name JOHN from byte 0, the starting character position is 1 or the letter J. If we READ from byte 4, the starting character position is 5 or the RETURN character. And so on.

An INPUT statement enters all characters beginning from the byte number specified to the next RETURN character. Continuing our example with

```
20 PRINT D$; "READ TEL LIST, B2"
30 INPUT N$
40 PRINT N$
```

line 40 prints HN from JOHN. That is, it begins at letter H (character position 3 or byte 2), and continues to the next RETURN character (at byte 4). To PRINT the entire name JOHN, we must READ from byte 0 as in

```
20 PRINT D$;"READ TEL LIST, B0"
```

If we were to incorrectly type

```
20 PRINT D$;"READ TEL LIST, B4"
```

then nothing will be printed to the screen, since byte 4 is the RETURN character.

Unlike POSITION, which can only move the pointer forward through a file, READ with a byte number can move it both forward and backward.

WRITING A BYTE

The form for WRITEing a specific byte is

WRITE F [,Bb]

where **F** is the file name and **b** is an absolute byte position.

This command instructs the computer to begin replacing characters at the specified byte with characters in the next PRINT statement. With reference to Fig. 9-12, if you write

```
20 PRINT D$;"WRITE TEL LIST, B2"
30 PRINT "AN"
```

the letters A and N will overwrite the third and fourth characters (H and N). You must confine overwriting to the current field. If you don't, problems will occur.

Let's look at what happens when you underwrite (i.e., have too few characters to overwrite a field). For example, the word JOHN* is 5 bytes long (including the RETURN character). Lines

```
20 PRINT D$;"WRITE TEL LIST, B1"
30 PRINT "A"
```

change the file from that shown in Figure 9-13 to that shown in Figure 9-14.

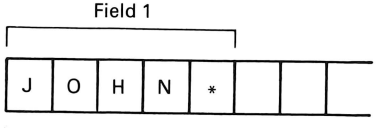


Figure 9-13 Field.

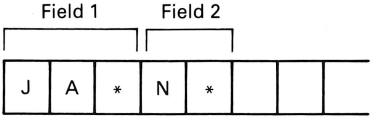


Figure 9-14 Underwriting.

Note that you have now created two separate fields.

On the other hand, if you overwrite with more characters than are in the field, as in

```
20 PRINT D$;"WRITE TEL LIST, B1"
30 PRINT "ALOPY"
```

the file changes from that shown in Figure 9-15 to that in Figure 9-16.

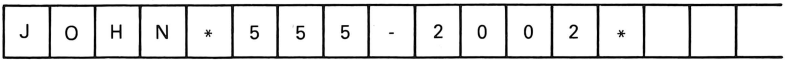


Figure 9-15 Sequential Format.

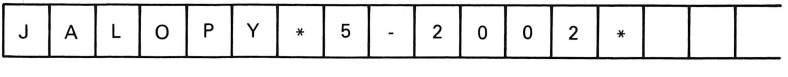


Figure 9-16 Overwriting.

Note that you have overwritten another field, destroying part of the data. You must obviously be more careful when WRITEing bytes than when READing them.

10 Random-Access Text Files

INTRODUCTION

In the last chapter we learned that sequential text files use disk space efficiently but access records slowly, and changes are difficult to make. In this chapter we will discuss random-access text files. These files access records quickly and changes are easy to make, but they don't conserve disk space very well.

The APPLE DOS MANUAL describes the random-access text file as like a honeycomb of equal-size cells. Some cells are full, some empty. Each cell is called a ***record.*** See Figure 10-1.

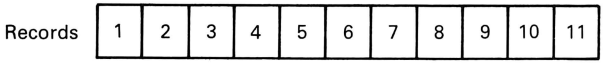


Figure 10-1 Record Format.

Each record contains one complete *data set*. A data set, as described in the last chapter, is a collection of data items. For example, a data set in a telephone list file could contain

Name
Telephone number
Address
City
State
Zip code

Before we can create a random-access text file, we need to define the record length, in bytes. Each record will then be that length, even if some bytes are empty.

In Chapter 8, we discussed briefly how to determine the storage requirements for string information. To review, each character in the string requires one byte of storage, plus one byte for "overhead" (the RETURN character). For example, the string "Sybil" is 6 bytes long (* = RETURN character). See Figure 10-2.

The numeric variable 123.41 uses seven bytes of storage. See Figure 10-3.

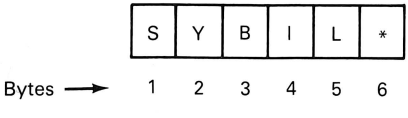


Figure 10-2 Number of Bytes.

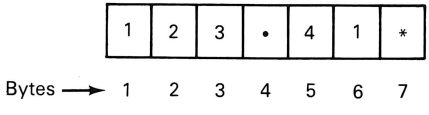


Figure 10-3 Number of Bytes.

The record size is based on the data set it contains. For illustration purposes, let's create a data set in a program that contains five data items.

```
10 INPUT "ENTER NAME: ";N$
20 INPUT "ENTER ADDRESS: ";AD$
30 INPUT "ENTER CITY: ";CT$
40 INPUT "ENTER STATE: ";ST$
50 INPUT "ENTER ZIP: ";ZP$
```

Our data set consists of the string variables N$,AD$,CT$,ST$, and ZP$.

Since a record within the file is a fixed length, we must establish a maximum length for each type of entry requested. We can rewrite our program to include this as follows:

```
10 INPUT "ENTER NAME (20 CHAR MAX): ";N$
20 INPUT "ENTER ADDRESS (25 CHAR MAX): ";AD$
30 INPUT "ENTER CITY (15 CHAR MAX): ";CT$
40 INPUT "ENTER STATE (2 CHAR MAX): ";ST$
50 INPUT "ENTER ZIP (5 CHAR MAX): ";ZP$
```

Adding together the maximum number of characters allowed for each string, including the "overhead byte," gives the record length.

Name	20 + 1 overhead = 21
Address	25 + 1 overhead = 26
City	15 + 1 overhead = 16
State	2 + 1 overhead = 3
Zip	5 + 1 overhead = 6
	Record length = 72

Each record in this data set will use 72 bytes of disk storage space, whether or not we completely fill the record.

If a data set uses only 50 bytes of the 72 allowed, then the remaining 22 bytes are wasted. You can easily see why random-access text files are not space-efficient. But, because each record is exactly the same size as all the others, you can access the last record in the file as quickly as the first.

The length of a record can be as small as 1 byte long, or as large as 32,767 bytes.

The procedure to create a random-access text file and retrieve information is similar to that of a sequential text file. The main difference is that certain commands have additional parameters. OPEN requires a **L** (length of record) parameter, whereas READ and WRITE each use a **R** (record number) parameter.

HOW TO CREATE A RANDOM-ACCESS TEXT FILE

We can summarize the process in four steps:

OPEN
WRITE
PRINT
CLOSE

To demonstrate how this works, let's create an inventory list. We will call our file **INV.** Each of the four steps will present a portion of the total program.

Since sequential and random-access text files are similar in construction, I will not repeat a lot of the information presented in Chapter 9. If you have not read Chapter 9, do so before proceeding.

Our program begins as follows:

```
 10 REM INVENTORY LIST
 20 :
 30 REM REST OF INTRODUCTION MODULE (see Chapter 8)
 40 :
    :
    :
    :
140 D$ = CHR$(4): REM CTRL-D
```

Let's use the following data set:

PART NUMBER	3 bytes + 1 overhead =	4
PRODUCT DESCRIPTION	15 bytes + 1 overhead =	16
QTY ON HAND	4 bytes + 1 overhead =	5
	Total bytes →	25

Our program lines are as follows:

```
160 HOME
170 INPUT "ENTER PRODUCT NUMBER (3 CHAR): ";PN$
180 REM ERROR TRAPS (see Chapter 6)
190 INPUT "ENTER PRODUCT DESCRIPTION (15 CHAR): ";PD$
200 REM ERROR TRAPS
210 INPUT "ENTER QUANTITY ON HAND (4 CHAR): ";QT
220 REM ERROR TRAPS
```

Step 1—Open

After determining the record length of the text file, your first step is to OPEN it. The OPEN statement opens the file, assigns a buffer, and indicates the record length in bytes. The format is

OPEN **f, Lj [,Ss] [,Dd] [,Vv]**

where **f** is the file name

L is the *L*ength-of-record

j is the number of bytes allotted to each record.

Ss, Dd, Vv, which stand for **S**lot number, **D**rive number, and **V**olume number, are optional.

In our program, the line is

```
300 PRINT D$;"OPEN INV, L25"
```

Be sure to include within the quotation marks the comma preceding L25. It is a part of the OPEN statement.

Note: When OPENing a random-access text file you must ***always*** indicate a ***L***ength of record, i.e., **L**j

Steps 2 and 3—Write and Print

The format for writing data to a random-access text file is

WRITE f [,Rr]

where **f** = file name
R = *R*ecord
r = the record number
We write to our file as follows:

```
310 PRINT D$;"WRITE INV, R1"
320 PRINT PN$: PRINT PD$: PRINT QT
330 PRINT D$
```

The WRITE command instructs the internal pointer to move to the Record number indicated. In line 310 we have instructed the computer to WRITE to Record number 1 (**,R1**). Typing

```
310 PRINT D$;"WRITE INV, R24"
```

would move the internal pointer to Record 24 to WRITE the strings following the PRINT statements in line 320. As in the OPEN statement, the comma that precedes **R1** is essential.

Line 320 PRINTs the indicated string information to the text file named INV on the disk.

Note: Never WRITE more bytes to a record then specified by the L parameter. This may partiallay overwrite records and produce confusing results.

The stand-alone PRINT D$ in line 330 turns off the WRITE process to the disk.

We should now consider two things:

1. Do we want to add more records?
2. How do we keep track of the number of records in our files?

First, to add more records, we need a loop, as in the sequential text file. Let's add the following lines:

```
350 PRINT "MORE ENTRIES? (Y/N): ";Y$
360 GET Y$: PRINT Y$
365 IF Y$ < > "Y" AND Y$ < > "N" THEN CALL -198: PRINT
    "ENTER "Y'" or "N": GOTO 350
370 IF Y$ = "N" THEN 500
380 :
390 :
400 GOTO 160
    :
    :
    :
500 REM CLOSE FILES
```

Line 350 asks for additional entries.

Line 365 error traps.

Line 370 sends the program to the CLOSE file routine if there are no more entries.

Line 400 sends the program to line 160 for additional data entry.

The second consideration is keeping track of the number of records in the text file. With sequential files, we aren't so concerned with how many records the file has. A loop READs through a file until an END OF DATA error occurs. ONERR GOTO then sends the program automatically to the CLOSE section.

Random-access text files must be treated differently. We need to keep track of how many records are in the file at all times. We use a counting statement such as **LET C = C + 1** for this. Let's set a default value of 1 to **C** and include it in our program.

```
150 LET C = 1
```

Entering the data requested in lines 160–220, and WRITEing it to the file in lines 300–330 creates a data set for one record. If there are no more entries, the program goes from line 370 to the file routine following line 500. The default value of 1 for the record count variable is correct.

To enter more data, we need

```
390 LET C = C + 1
```

Line 390 continues to increase the record count (variable C) by 1 each time more data is added. When data entry is complete, line 370 skips line 390 and goes to the CLOSE routine.

The current value of **C** should be stored in the file for future use. **R0** (Record zero), the first record in the file, is a convenient place. (You probably thought **R1** was the first record in the file!)

Step 4—Close

To save the new value of **C** (the number of records we have written) and CLOSE the file, we add these lines:

```
500 REM CLOSE FILE
510 PRINT D$; "WRITE INV, R0"
520 PRINT C
530 PRINT D$
540 PRINT D$; "CLOSE"
550 PRINT "FILES UPDATED"
560 END
```

You must ***always*** CLOSE a file when you're through with it. At this point, our program looks like this:

```
10 REM INVENTORY LIST
20 :
30 REM REST OF INTRODUCTION MODULE
```

```
 40 :
    :
    :
    :
140 D$ = CHR$(4): REM CTRL-D
150 LET C = 1
160 HOME
170 INPUT "ENTER PRODUCT NUMBER (3 CHAR): ";PN$
180 REM ERROR TRAPS
190 INPUT "ENTER PRODUCT DESCRIPTION (15 CHAR): ";PD$
200 REM ERROR TRAPS
210 INPUT "ENTER QUANTITY ON HAND (4 CHAR): ";QT
220 REM ERROR TRAPS
    :
    :
300 PRINT D$;"OPEN INV, L25"
310 PRINT D$;"WRITE INV, R1"
320 PRINT PN$: PRINT PD$: PRINT QT
330 PRINT D$
350 PRINT "MORE ENTRIES? (Y/N): ";Y$
360 GET Y$: PRINT
365 IF Y$ < > "Y" AND Y$ < > "N" THEN CALL -198: PRINT
    "ENTER 'Y' OR 'N'": GOTO 350
370 IF Y$ = "N" THEN 500
380 :
390 LET C = C + 1
400 GOTO 160
    :
    :
    :
500 REM CLOSE FILE
510 PRINT D$;"WRITE INV, R0"
520 PRINT C
530 PRINT D$
540 PRINT D$;"CLOSE"
550 PRINT "FILES UPDATED"
560 END
```

We will discuss in the next section how to use the information stored in Record zero. But before we do, let's cover a few more things.

You may find that a filename is quite long and, therefore, inconvenient to type. Buy you may not want to shorten it and risk forgetting what the file contains.

For example, what if your file is called UNIV SO CALIF? It might appear in a program as

```
PRINT D$;"OPEN UNIV SO CALIF,L40"
PRINT D$;"WRITE UNIV SO CALIF,R5"
PRINT D$;"READ UNIV SO CALIF,R10"
PRINT D$;"CLOSE UNIV SO CALIF"
```

You might be tempted to shorten the file name to USC. Later, however, you may not be sure what the file contains. Is it information on the University of Southern California, South Carolina, or Southern Colorado?

One way programmers handle long filenames is to set the filename to a string variable, e.g.,

```
20 F$ = "UNIV SO CALIF"
```

A string variable is easier to use, faster to type, and keeps typing errors to a minimum. In a program it looks like

```
 10 REM UNIV SO CALIF
 20 :
 30 REM REST OF INTRODUCTION MODULE (see Chapter 8)
 40 :
    :
    :
    :
130 F$ = "UNIV SO CALIF"
140 D$ = CHR$(4)
150 :
160 REM DATA ENTRY MODULE
    :
    :
    :
300 PRINT D$;"OPEN ";F$;",L40"
310 PRINT D$;"WRITE ";F$",R4"
320 :
330 :
340 PRINT D$;"CLOSE";F$
```

Now every time you type F$ the program will expand it to UNIV SO CALIF. We can make this process even easier. The DOS commands OPEN, WRITE, READ, and CLOSE occur several times during a large program. Why not set the DOS commands and filename together to a string variable as well? We can use the following method:

```
110 OP$ = "OPEN UNIV SO CALIF, L40"
115 WR$ = "WRITE UNIV SO CALIF, R"
120 RD$ = "READ UNIV SO CALIF, R"
125 CL$ = "CLOSE UNIV SO CALIF"
```

We should also change several lines in our previous program.

```
300 PRINT D$;OP$
310 PRINT D$;WR$;5
320 PRINT D$;CH$
```

Line 300 OPENs the file. Line 310 WRITEs to Record 5. Line 320 CLOSEs the file.

We can go still further:

```
 10  D$ = CHR$(4)
 20  OP$ = D$ + "OPEN UNIV SO CALIF, L40"
     :
     :
     :
     :
300  PRINT OP$
```

RETRIEVING INFORMATION FROM A RANDOM-ACCESS TEXT FILE

Retrieving information from a random-access text file is the same as from a sequential text file with the exception of adding the R (Record number) parameter. The process can be summarized in four steps:

OPEN
READ
INPUT
CLOSE

Step 1—Open

This step is the same as the one we used in writing to the file.

```
 10  REM INVENTORY LIST
 20  :
 30  REM REST OF INTRODUCTION MODULE
 40  :
     :
     :
     :
100  D$ = CHR$(4):REM CTRL-D
110  PRINT D$;"OPEN INV, L25"
```

It is extremely important that the retrieval Record *L*ength be the same as for writing to the file. Otherwise, "DOS will blindly use the *new* Length parameter to calculate record positions within the file."[1] Using any other L parameter produces confusing results.

[1]Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

Steps 2 and 3—Read and Input

We mentioned earlier that it is very important to keep track of how many records we have in the file. When we wrote to file INV, we stored the record count in Record zero, the first record in the file. It is now time to use that data in the following lines.

```
120 PRINT D$;"READ INV, R0"
130 INPUT C
140 PRINT D$
```

Line 120 instructs the pointer to move to **R0** (Record zero).

Line 130 INPUTs the record count and sets it to the numeric variable C.

Line 140 stops the INPUT process from the disk.

Now that we know exactly how many records are in file, it is easy to create a FOR NEXT loop to INPUT all the data.

```
145 PRINT "PROD #";TAB (10);"PROD DESCRIP";TAB (18);"QUANTITY"
150 :
160 REM READ FILE
170 :
180 FOR X = 1 TO C
190 PRINT D$;"READ INV, R";X
200 INPUT PN$,PD$,QT
210 PRINT D$
220 PRINT PN$;TAB (10);PD$;TAB(20);QT
230 NEXT X
240 :
```

Line 145 creates a heading for the data.

Lines 180–230 create a FOR NEXT loop to INPUT C records and display the data.

Note the format of line 190. We have maintained the necessary elements for a READ statement. The **,R** is still within quotation marks. The value of **X** (outside the quotation marks) changes with each FOR NEXT loop.

We have avoided using the record count in ONERR to determine the end of the file. The record count must be accurate. If it is one short, it omits data. A count of one too many causes an END OF DATA error.

Step 4—Close

The fourth step is the same as before.

```
250 REM CLOSE FILE
260 :
270 PRINT D$;"CLOSE"
280 END
```

Our final program looks like this:

```
 10 REM INVENTORY LIST
 20 :
 30 REM REST OF INTRODUCTION MODULE
 40 :
    :
    :
    :
100 D$ = CHR$(4): REM CTRL-D
110 PRINT D$;"OPEN INV, L25"
120 PRINT D$;"READ INV, R0"
130 INPUT C
140 PRINT D$
145 PRINT "PROD ";TAB (10);"PROD
    DESCRIP";TAB (18);"QUANTITY"
150 :
160 REM READ FILE
170 :
180 FOR X = 1 TO C
190 PRINT D$;"READ INV, R";X
200 INPUT PN$,PD$,QT
210 PRINT D$
220 PRINT PN$;TAB (10);PD$;TAB(20);QT
230 NEXT X
240 :
250 REM CLOSE FILE
260 :
270 PRINT D$;"CLOSE"
280 END
```

HOW TO ADD INFORMATION TO A RANDOM-ACCESS TEXT FILE

So far, we have discussed only saving the record count in **R0** (Record zero) when retrieving data. The record count is also important when one is writing to a file.

Setting the record count variable C to a default value of 1 does not take existing records into account. That is, RUNning the program sets C automatically back to the default value each time, regardless of the number of records in the file.

For example, if the file contains 20 records and you plan to enter four more, C should be equal to 20.

We can easily correct this problem by replacing the default statement C = 1 with the following lines:

```
130 INPUT "ENTER FILE NAME: ";F$
140 PRINT D$;"OPEN ";F$;",L25"
150 PRINT D$;"READ ";F$;",R0"
160 INPUT C
170 PRINT D$
180 LET C = C + 1
```

Now the program READs the current number of records and assigns that number to the variable C. Line 180 increases the record count by 1.

Now we must ask. What happens if we enter a filename that does not exist on the disk?

The answer is that line 140 OPENs the file and assigns a buffer, and lines 150–160 try to READ data from it.

This results in an END OF DATA error, and the program stops abruptly. We obviously need a way to first check whether the file exists on the disk. If it doesn't, the program should store the newly created file's record count as zero.

We can do this with **VERIFY** and **ONERR.**

```
135 ONERR GOTO 1000
137 PRINT D$;"VERIFY ";F$
139 POKE 216,0


1000 REM ERROR TRAP
1010 IF PEEK(222) < > 6 then 1100
1020 CALL -198: PRINT "FILE ";F$;" DOES NOT EXIST."
1030 PRINT "DO YOU WANT TO CREATE (Y/N)?";
1040 GET Y$ :PRINT Y$
1050 IF Y$ = "N" THEN 130
1060 PRINT D$;"OPEN ";F$;",L25"
1070 PRINT D$;"WRITE ";F$;",R0"
1080 PRINT 0
1090 PRINT D$;"CLOSE"
1095 GOTO 140
1100 PRINT "UNUSUAL ERROR: ";PEEK (222)
1110 END
```

Line 137 verifies the file. If it doesn't exist, an error occurs.

Line 135 traps the error and sends the program to line 1000.

Line 1010 checks whether the error code is 6, FILE NOT FOUND. It sends any other error code to line 1100, which displays it and ends the program.

Lines 1020–1050 let you create a new file. If you mistyped the file name, line 130 lets you reenter it.

Lines 1060–1095 create the new file with a record count of 0 (zero). Then the program returns to line 140.

More on R0

Let's discuss another potential problem and its solution when one is working with **R0.** Suppose that you want to create a billing program. You decide that you will store each month's invoices in a separate file, for example,

MONTH 1
MONTH 2
MONTH 3

and so on.

Your first step is to determine what each data set should consist of. Next, compute the record length. Let's assume it is 205 bytes.

You then decide to create all the text files at once, as follows:

```
10 REM CREATE FILES
20 D$ = CHR$ (4)
30 FOR M = 1 TO 12
40 PRINT D$;"OPEN MONTH ";M;",L205"
50 PRINT D$;"WRITE MONTH ";M;",R0"
60 PRINT 0
70 PRINT D$;"CLOSE"
80 NEXT M
90 END
```

This program creates each month's files and stores the record count of 0 in R0 (Record zero).

Let's see what happens if we use our Retrieve Information program on a file that doesn't contain any records.

```
100 INPUT "ENTER MONTH TO READ: ";M
120 PRINT D$;"OPEN MONTH ";M;",L205"
140 PRINT D$;"READ MONTH ";M;",R0"
160 INPUT C
180 PRINT D$
185 :
190 REM READ DATA MODULE
195 :
200 FOR X = 1 TO C
```

```
220 PRINT D$;"READ MONTH ";M;",R";X
240 :
250 :
260 REM INPUT DATA MODULE
280 :
```

Line 160 INPUTs the number of records in file.

If C = 0, then there are no records in file. When the program continues to lines 200 and following, an END OF DATA error occurs. We need an error trap to bypass the READ DATA MODULE and advise the user that there are no records in the file. We add the following lines:

```
170 IF C = 0 THEN PRINT D$;"CLOSE": CALL -198: PRINT "NO RECORDS
IN FILE.": GOTO 10
```

Line 170 provides an error trap for no record in file, closes the file, and advises the user. Then the program goes back to the beginning to let the user try again or exit.

HOW TO MAKE CHANGES IN A RANDOM-ACCESS TEXT FILE

Because a random-access text file is divided into equal-size records, we can access a particular record quickly and make changes easily. Each record is like a small sequential text file.

Let's use our INVentory program to demononstrate how to make changes in the records. The program assumes that the parts are numbered in order, from 1 on.

```
 10 REM INVENTORY
 20 D$ = CHR$(4)
 30 HOME
 40 OP$ = D$ + "OPEN"
 50 WR$ = D$ + "WRITE"
 60 RD$ = D$ + "READ"
 70 CL$ = D$ + "CLOSE"
 80 F$ = "INVENTORY"
100 PRINT OP$;F$;",L25"
110 PRINT RD$;F$;",R0"
120 INPUT C
130 PRINT D$
135 GOSUB 500: REM MAIN MENU
140 ON N GOTO 150,250,700
150 GOSUB 615: REM INPUT PART #
160 FOR I = LOW TO HIGH
170 PRINT RD$;F$;",R";I
180 INPUT PN$,PD$,QT
```

```
190 PRINT "   ";PN$;TAB (10);PD$;TAB(28);QT
200 NEXT I
210 PRINT D$
220 GOTO 430
230 :
240 REM CHANGE INFO
250 VT = 4: FOR I = LOW TO HIGH: VT = VT + 1
260 PRINT RD$;F$;",R";I
270 INPUT PN$,PD$,QT
280 PRINT D$
290 VTAB (VT):PRINT "   ";PN$;: HTAB 1: INPUT S$
300 IF LEN (S$) > 3 THEN 260
310 IF LEN (S$) < > 0 THEN PN$ = S$
320 VTAB (VT): HTAB 10: PRINT PD$;:HTAB 9: INPUT S$
330 IF LEN (S$) > 15 THEN 320
340 IF LEN (S$) < > 0 THEN PD$ = S$
350 VTAB (VT): HTAB 28: PRINT QT;:HTAB 27: INPUT S$
360 IF LEN (S$) > 4 THEN 350
370 IF LEN (S$) < > 0 THEN QT = VAL (S$)
380 VTAB (VT): PRINT "   ";PN$;TAB (10);PD$;TAB(28);QT
390 PRINT WR$;F$;",R";I
400 PRINT PN$: PRINT PD$: PRINT QT
410 PRINT D$
420 NEXT I
430 VTAB (23): PRINT "PRESS THE RETURN KEY TO CONTINUE.";:GET Y$
440 GOTO 135

500 REM MENU
510 HOME
520 PRINT TAB (20 - (LEN (F$)/2)) F$
530 PRINT
540 PRINT "COMMAND OPTIONS:"
550 PRINT
560 PRINT "1) LIST"
570 PRINT "2) CHANGE"
580 PRINT "3) EXIT":PRINT
590 INPUT "SELECT NUMBER (1-3): ";N$: LET N = VAL (N$)
600 IF N > 0 AND N < 4 THEN RETURN
610 CALL -198: GOTO 510

615 REM INPUT PART #
618 PRINT
620 INPUT "ENTER PART NUMBER (0=ALL): ";N$
630 N = VAL (N$): IF N < 0 THEN CALL -198: GOTO 620
640 LOW = N: HIGH = N
650 IF N = 0 THEN LOW = 1: HIGH = C: REM C=# OF RECORDS IN FILE
660 HOME
670 PRINT "PART #";TAB(10);"PROD DESCRIP";TAB (28);"QUANTITY"
680 PRINT "-----   ";TAB(10);"---- -------";TAB (28);"--------"
```

```
690 RETURN
700 PRINT CL$;F$
710 HOME: END*
```

Lines 10–80 assign the variables

Lines 100–130 open the INV file and read the number of records.

Line 135 sends the program to a subroutine to display the main menu.

Lines 150–220 ask which data to display and display it.

Lines 240–440 allow changes to the data.

Lines 500–610 display menu options.

Lines 615–690 prompt for part number.

Lines 700–710 CLOSE the file

Let's discuss lines 240–690 in more detail.

Line 250 creates a FOR NEXT loop to READ through the file's data. The variables LOW and HIGH were established earlier in the subroutine in lines 615–690.

Lines 615–690 let the user determine which records to READ. If only one record will be changed, the variables **LOW** and **HIGH** will be the same. If all the data will be changed, LOW will be equal to 1 and HIGH will be equal to the variable C (representing the number of records in file, assigned in line 120).

A counting routine, **VT = VT + 1,** is also established. **VT** will control the VTAB line in which each part number, description, and quantity will appear.

Line 260 READs the record number I (determined by LOW to HIGH).

Line 270 INPUTs the three variables.

Line 280 stops the INPUT process.

Line 290 presents an interesting technique. VTAB (VT) establishes the text screen line on which the data will appear. The first time through the loop, VT is equal to 4 (assigned in line 250).

Each item is displayed at a particular HTAB position.

A Quick Review

An INPUT statement containing a message, for example,

```
10 INPUT "ENTER NAME: ";N$
```

produces a blinking cursor. See Figure 10-4.

A stand-alone INPUT statement, as in

```
10 PRINT "NAME: ";
20 INPUT N$
```

produces a question mark followed by the blinking cursor.

*Based on "RANDOM" program from APPLE II SYSTEM MASTER.

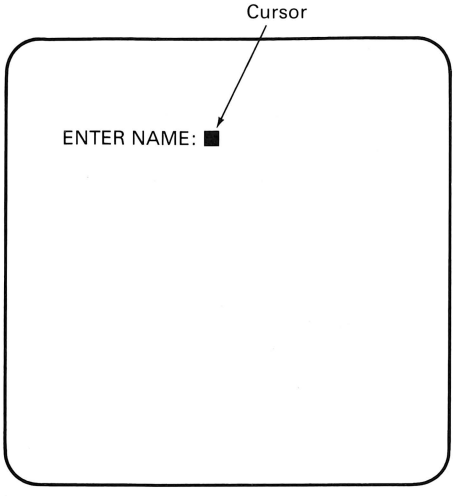


Figure 10-4 Screen Display with Cursor.

Following each PRINT statement with **HTAB −1** and a stand-alone INPUT statement makes a question mark appear one space before the item printed. The blinking cursor appears over the first character of the item. You can then either change the data or press <RETURN> to continue. If you don't change anything, the program skips lines 300 and 310 and displays the next item (in line 320).

If you do change the data, the new entry is temporarily assigned to string S$. Line 310's condition is met and N$ is now changed to equal the new item defined by S$.

Line 300 checks that the length of the new entry does not exceed the maximum length allowed for the string.

Lines 320–370 repeat the process.

Line 380 displays the entire data set.

Lines 390–410 reWRITE the data to the same Record number I (determined by LOW to HIGH).

Line 420 completes the FOR NEXT loop and returns, if instructed, to INPUT the next record.

DELETE RANDOM-ACCESS TEXT FILES

Random-access text files are deleted in the same way as sequential text files. Refer to Chapter 9 for a full explanation.

COPYING A RANDOM-ACCESS TEXT FILE

You can copy a random-access text file directly to another disk by using the FID program on your APPLE SYSTEM MASTER diskette. Refer to Chapter 9 for a full explanation.

Another Way To Copy

You may also write a program to copy the entire contents of one file to another. Let's use our inventory program INV as an example and copy the data to a second file called INV.2.

```
 10 REM COPY PROGRAM
 20 :
    :
    :
 40 REM INTRODUCTION MODULE
    :
    :
    :
    :
100 D$ = CHR$(4)
120 PRINT D$;"OPEN INV, L25"
130 PRINT D$;"OPEN INV.2"
140 PRINT D$;"DELETE INV.2"
150 PRINT D$;"OPEN INV.2,L25"
160 :
170 REM READ FROM SOURCE FILE
180 :
190 PRINT D$;"READ INV, R0"
200 INPUT C1
210 PRINT D$
220 FOR J = 1 TO C1
230 HOME: FLASH: PRINT "READING": NORMAL
240 PRINT D$;"READ INV,R";J
250 INPUT PN$,PD$,QT
260 PRINT D$
270 :
280 REM COPY TO FILE
290 :
300 HOME: FLASH: PRINT "COPYING": NORMAL
310 PRINT D$;"WRITE INV.2,R";J
320 PRINT PN$:PRINT PD$: PRINT QT
330 PRINT D$
340 NEXT J
350 :
360 REM CLOSE FILES
370 PRINT D$;"WRITE INV.2,R0"
380 PRINT C1
390 PRINT D$;"CLOSE"
400 HOME: PRINT "COPY FININSHED"
410 END
```

Lines 10–100 are the introduction module.

Lines 120–150 OPEN the INV and INV.2 files. Note that we omitted ",L25" in line 130. Since we plan to DELETE the file in line 140, we do need to include the Length in line 150.

Lines 190–210 READ the number of records in INV.

Lines 220–260 INPUT data from INV.

Lines 300–330 copy data to INV.2.

Lines 370–410 update the record count in the new file, INV.2, and CLOSE both files.

In Chapter 9 we modified the Copy program to let us select data sets in a sequential text file and move them to another file. The procedure is similar for random-access text files.

The only difference is that it must keep an accurate count of the records stored in **R0** (record zero) after a transfer. We can do this by adding a counting statement to the previous program.

```
110  C2 = 1
      :
      :
      :
262  HOME: PRINT PN$;TAB(10);PD$;TAB (28);QT
264  PRINT:PRINT
266  PRINT "DO YOU WISH TO COPY (Y/N): ";
268  GET Y$: PRINT
270  IF Y$ = "N" THEN 340
275  C2 = C2 + 1
```

Change line 380 to:

```
380  PRINT C2
```

ALPHABETIZING A RANDOM-ACCESS TEXT FILE

Alphabetizing data retrieved from any text file requires a sort routine (usually a subroutine). It comes ahead of displaying the data or SAVEing it to another text file. See Chapter 7 for some sort routines.

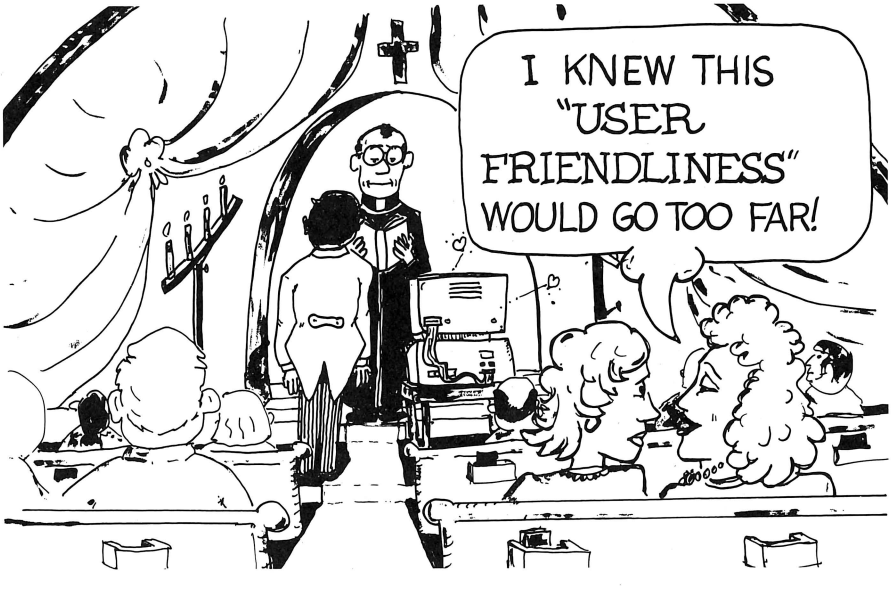
SEARCHING FOR SPECIFIC DATA

We can use ***keywords*** to find specific data in a file. A keyword can be anything distinctive in the data you want, such as a part number, a name, etc. When the computer finds a match, it displays the data.

Let's say that you have an inventory program with a file containing a 4-digit part number (PN), a product name (PD$), and the quantity on hand (QT). Suppose you want to find out how many whiz-bangs you have in stock. The part number of 2436.

The following search program uses the part number as the keyword.

```
 10 REM SEARCH PROGRAM
 20 D$ = CHR$(4)
    :
    :
 40 REM REST OF INTRODUCTION MODULE
    :
    :
    :
    :
100 HOME
110 PRINT TAB (12) "INVENTORY SEARCH"
120 PRINT:PRINT
130 INPUT "ENTER PART NUMBER: ";N$
135 LET N = VAL (N$)
140 If N < = 0 THEN CALL -198: PRINT:PRINT "PART NUMBER CANNOT BE 0 OR
    LESS.": GOTO 130
150 IF LEN (N$) < > 4 THEN CALL -198: PRINT "PART NUMBER MUST BE 4 DIGITS.":
    PRINT "PLEASE REENTER.": GOTO 130
160 PRINT:PRINT
170 FLASH: PRINT "PLEASE WAIT WHILE SEARCHING": NORMAL
```



```
180 PRINT D$;"OPEN INV, L25"
190 PRINT D$;"READ INV, R0"
200 INPUT C
210 PRINT D$
220 FOR J = 1 TO C
230 PRINT D$;"READ INV,R";J
240 INPUT PN
250 IF N < > PN THEN 330
260 INPUT PD$,Q
270 PRINT D$;"CLOSE"

280 REM MATCH FOUND!
290 HOME
300 POKE 34,5
310 PRINT PN; TAB(10);PD$;TAB(28);Q
320 GOTO 350

330 NEXT J

300 REM NO MATCH FOUND
310 PRINT D$;"CLOSE"
320 HOME
330 CALL -198: PRINT "UNABLE TO LOCATE PART NUMBER ";N

350 VTAB 23: PRINT "SEARCH AGAIN? (Y/N): ";
360 GET Y$:PRINT: IF Y$ = "N" THEN 400
370 POKE 34,0: GOTO 100
400 POKE 34,0:HOME:END
```

The key to this program is line 250. It compares the part number entered (in line 130) with the first entry (PN) in the record. If they don't match, the program passes over all the rest of the data in that record and skips to the next record.

We could just as easily have used Product description (PD$) as the keyword. This would require the following changes in the program.

```
130 INPUT "ENTER PRODUCT DESCRIPTION: ";P$
140 IF LEN (P$) < 1 or LEN (P$) > 15 THEN CALL -198: VTAB
    PEEK(37): GOTO 130
150 :
    :
    :
    :
    :
240 INPUT PN
245 INPUT PD$
250 IF P$ < > PD$ THEN 290
260 INPUT Q
```

If you intend to use a search feature in a program, think carefully about how to organize the variables. Putting a keyword variable near the top of a record speeds up searches.

BYTE CONTROL

In this chapter, we have discussed how to change data in a record. Suppose, for example, we want to change the quantity on hand in record 5 in the inventory program. We could summarize the procedure as follows:

1. INPUT *all* data items in record 5, e.g.,

```
20 INPUT PN, PD$, QT
```

2. Change the variable QT to reflect the new value.
3. WRITE *all* the data items back to record 5, e.g.,

```
100 PRINT PN: PRINT PD$: PRINT QT
```

Note that the procedure requires *all* variables to be INPUT and rewritten. The purpose, of course, is to avoid overwriting or underwriting a file. See Chapter 9 for a discussion of overwriting and underwriting files.

We could use the Byte parameter (see "Byte Control" in Chapter 9). But, as the APPLE DOS MANUAL states, "Using the B parameter necessitates a thorough, detailed, byte-by-byte knowledge of the contents of each record in the file."[2]

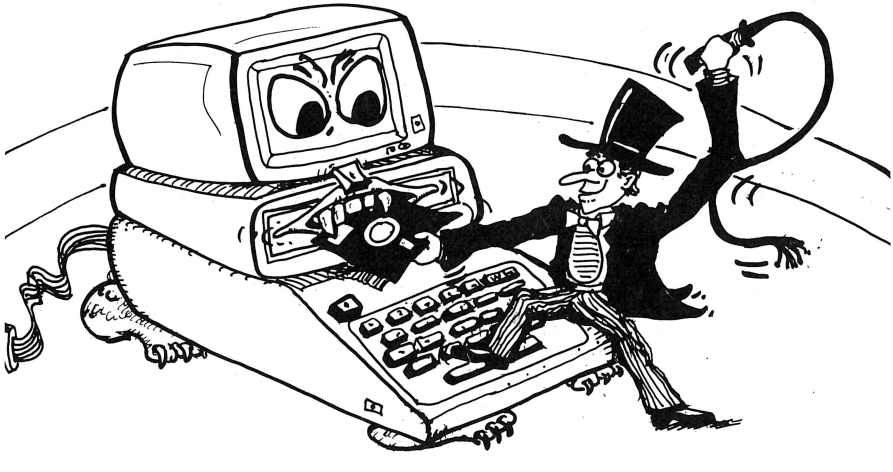


Figure 10-5 Byte Control.

[2]Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

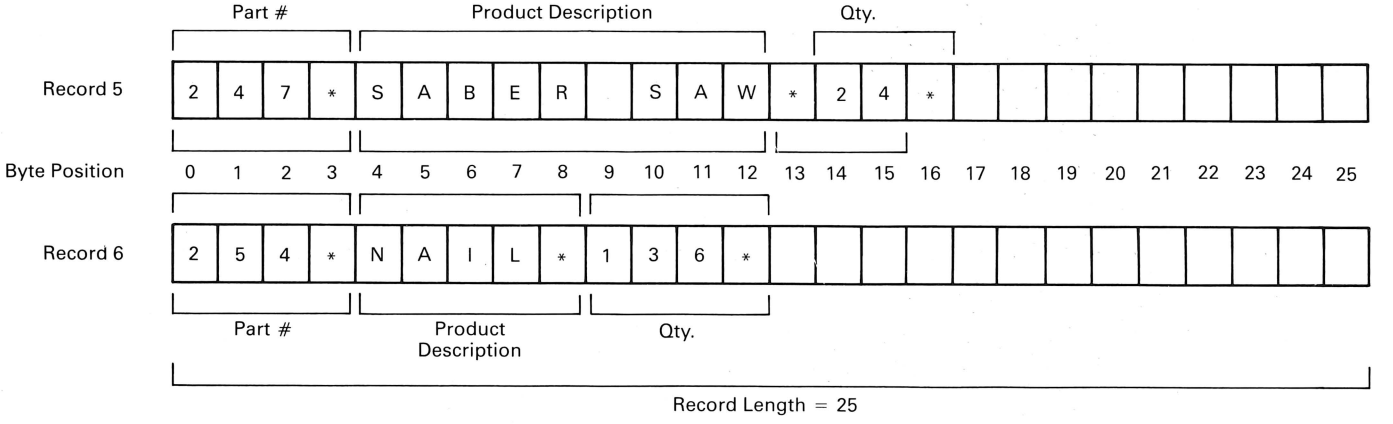

Part #
Product Description
Qty.
Record 5
2 4 7 * S A B E R S A W * 2 4 *
Byte Position
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Record 6
2 5 4 * N A I L * 1 3 6 *
Part #
Product Description
Qty.
Record Length = 25

Figure 10-6 Random-access File Use.

This would be difficult, since data items usually vary in length from one record to another. You can, however, construct a file to make using the B parameter reasonable. We know that each record of a random-access text file has a fixed length. But each record may not be completely filled with data. For example, see Figure 10-6.

The data in record 5 occupies 16 bytes. In record 6 it occupies 12 bytes. In record 5, QTY begins at byte 14. In record 6, it begins at byte 9.

Using the byte parameter is not practical when string lengths are variable within a record. Fortunately, we can make all items of a particular type begin at the same byte position in each record.

Padding a String

We can extend the length of an entry by ***padding*** it. By padding a short entry with spaces, we can make the string fit exactly in the data field. The following lines demonstrate this technique.

```
200 INPUT "ENTER PRODUCT DESCRIPTION (MAX 15 CHAR): ";PD$
210 IF LEN (PD$) > 15 THEN PRINT "ENTRY TOO LONG": GOTO 200
220 IF LEN (PD$) < 15 THEN LET PD$ = PD$ + " ": GOTO 220
```

Line 200 limits the characters to 15.

Line 210 finds entries that are too long.

Line 220 finds entries that are too short and keeps adding trailing spaces until PD$ is equal to 15 characters.

We must establish a maximum for each data field in the record to make the field following it start in a particular byte position. In the example in Fig. 10-7, all part numbers begin at byte 0, all product descriptions begin at byte 4, and all quantities begin at byte 20.

Now we can use the byte parameter to display only the product descriptions.

```
 90 PRINT D$;"OPEN INV,L25"
100 FOR J = 1 TO ( of records in file)
110 PRINT D$;"READ INV,R";C;",B4"
120 INPUT PD$
130 PRINT PD$
140 PRINT D$
150 NEXT J
160 :
    :
    :
```

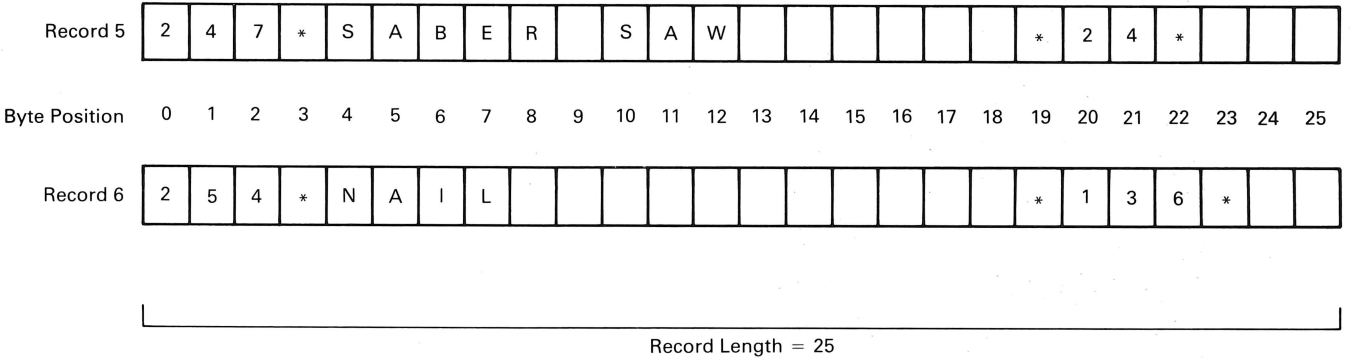


Figure 10-7 Random-access Padded Strings.

This is an easy way to print a specific data field from a record containing many data items. Writing to a specific data field is just as easy.

```
300 PRINT D$;"WRITE INV,R";J;",B4"
310 PRINT PD$: REM PAD PD$
320 PRINT D$
```

11 One Final Word

HOW TO GET FREE OR INEXPENSIVE SOFTWARE

Would you like to improve your programming skills? Perhaps even create your own programs? If so, you can get help at little or no cost.

Several magazines on the market are totally dedicated to the APPLE computer. They cover a broad spectrum of programming topics, including business, education, games, graphics, and utilities. Whatever application you're looking for, you'll probably find it in one of the magazines.

The following list describes several Apple-related magazines and the subscription address of each.

1. *A +*
   the Independent Guide
   for Apple Computers
   Ziff Davis Publishing Company
   One Park Ave
   New York, NY 10016

This is a general-interest magazine that covers education, graphics, software and hardware review, utility programs, etc. Published monthly.

2. *inCider*
The Apple II Journal
CW Communications/Peterborough, Inc.
80 Pine Street
Peterborough, NH 03458

An excellent magazine that covers a wide variety of topics. Special departments include Hint's 'n' Techniques, the Applesoft Advisor, Fudge It!, the Apple Clinic, plus much more. Plenty of programs and good articles! Published monthly.

3. *nibble*
The Reference for Apple Computing
MicroSPARC, Inc.
45 Winthrop Street
Concord, MA 01742

This magazine is packed with programs. Topics and features also include education, games, utilities, Tips 'n' Techniques, Assembly, Pascal, Legal bits, and special departments. A valuable magazine for any Apple user. Published monthly.

4. *CORE/HARDCORE Computist*
SoftKey Publishing
PO BOX 44549
Tacoma, WA 98444

"WE ARE NOT PIRATES" is their ad heading in magazines. *HARCORE* is for the computer user who wants to know how to unlock locked software. This is a controversial area. But SoftKey feels users should have the right to make back-up copies. *HARDCORE* is published eight times a year. *CORE,* which contains in-depth coverage of special-interest subjects, is published quarterly.

5. *BARON'S MICROCOMPUTING REPORTS*
Personalized Report for: APPLE
BARON'S MICROCOMPUTING REPORTS
344 E. 49th Street
New York, NY 10017

One of several personalized reports Baron publishes for different computers. This is a well-done 12–16-page report covering a broad area of topics, including hardware/software reviews, tips, etc. Check it out.

6. *CALL -A.P.P.L.E.*
Apple PugetSound Program Library Exchange
290 SW 43rd St.
Renton WA 98055

A.P.P.L.E. is the world's largest Apple User group (more information on them later). They publish their own magazine monthly. It is loaded with articles, reviews, education news, and departments. It's a magazine well worth subscribing to.

7. *IAC Disk of the Month*
International Apple Core, Inc.
Membership Dept. I–04
467 Saratoga Ave. Ste 228
San Jose, CA 95129

International Apple Core is a worldwide federation of Apple computer user groups. Each month they publish the IAC Disk of the month, a subscription to user contributed software. Each disk presents themes such as games, education, or a potpourri.

8. *SOFTDISK*
SOFTDISK
P.O. Box 30008
Shreveport, LA 71130-0008

SOFTDISK is a magazine on floppy disks. Each month two double-sided diskettes packed full of excellent programs are issued. The disks are unprotected allowing you to backup, modify, and learn from the programs.

A Source for Free Software

Throughout this book we credit several programs to the Public Domain Library. This library provides software free or for a nominal charge. Public domain software (PDS) is copyright-free. That means you can copy it or give it away without any fear of breaking the law. PDS gives you an opportunity to become involved in areas you might not otherwise have a chance to explore. PDS includes games, art and graphics, science, mathematics, aviation, utilities, business, tutorials, education, music, electronics, etc.

As you might expect, PDS does have disadvantages. It is often called "as is" software. This means that the programs aren't as powerful or as sophisticated as commercially available software. Some programs may contain bugs. Most work fine, however, and you can modify them to suit your own needs.

PDS is available from many sources. Some businesses offer PDS in magazine advertisements. Computer clubs often have a Public Domain Library in which you may purchase or copy a PDS disk. One such club is A.P.P.L.E. (Apple PugetSound Program Library Exchange).

Prices vary. All diskettes run under DOS 3.3. A catalog of listing is available for a nominal charge.

Programs are not guaranteed to run. They do not include documention, although some programs are self-documented. The price is well worth the investment.

COMPUTER USERS' GROUP

Joining an Apple users' group provides many benefits. Users' groups can be international in scope or just a local group of individuals who want to get together and have fun computing. One international group is A.P.P.L.E. (Apple PugetSound Program Library Exchange).

A.P.P.L.E. is now the world's single largest Apple computer users' group. It is headquartered in Renton, Washington, and has a membership of over 25,000. The service-oriented organization offers its members comprehensive support for Apple computers and personal computing in general.

Membership in A.P.P.L.E. CO-OP includes hotline privileges for technical assistance, the international magazine *CALL -A.P.P.L.E.*, major discounts on hardware and software products, classes within A.P.P.L.E.'s own educational facility, special-interest computer groups, bimonthly promotional events, and access to an A.P.P.L.E.-owned store. Since is it a co-operative, members share equally in profits. If you are interested in becoming a club member, write to

A.P.P.L.E.
290 SW 43rd ST.
Renton, WA 98055.

ACKNOWLEDGMENT

I would be remiss if I did not acknowledge a particular book that was of great benefit to me. Although listed in the bibliography, it deserves greater attention and noting within a chapter context.

The book is entitled *Apple Basic: Data File Programming* by LeRoy Finkel and Jerald R. Brown. The majority of my understanding in text file programming was gained through this book. It is the first book I have found that shows the finer points of text file programming.

Although I have covered a great deal of the same information discussed in their book, I highly recommend that you purchase it. Some functions are covered in greater detail. In addition, the authors have included several practical programs that use text files, such as business, personal finance, shopping lists, etc.

I would also like to thank all the fine folks at A.P.P.L.E., who allowed me to freely go through the Public Domain Library.

Peggy Roarty 68
850815-11
462–471

APPENDIX A
ASCII CHARACTERS

DEC = ASCII decimal value
n/a = not accessible directly from the APPLE II keyboard

DEC		*DEC*		*DEC*	
0	CTRL-@	32	space	64	@
1	CTRL-A	33	!	65	A
2	CTRL-B	34	"	66	B
3	CTRL-C	35	#	67	C
4	CTRL-D	36	$	68	D
5	CTRL-E	37	%	69	E
6	CTRL-F	38	&	70	F
7	CTRL-G	39	'	71	G
8	CTRL-H	40	(	72	H
9	CTRL-I	41	)	73	I
10	CTRL-J	42	*	74	J
11	CTRL-K	43	+	75	K
12	CTRL-L	44	,	76	L
13	CTRL-M	45	−	77	M
14	CTRL-N	46	.	78	N
15	CTRL-O	47	/	79	O
16	CTRL-P	48	0	80	P
17	CTRL-Q	49	1	81	Q
18	CTRL-R	50	2	82	R
19	CTRL-S	51	3	83	S
20	CTRL-T	52	4	84	T
21	CTRL-U	53	5	85	U
22	CTRL-V	54	6	86	V
23	CTRL-W	55	7	87	W
24	CTRL-X	56	8	88	X
25	CTRL-Y	57	9	89	Y
26	CTRL-Z	58	:	90	Z
27	ESC	59	;	91	[n/a
28	n/a	60	<	92	\ n/a
29	ctrl shift-M	61	=	93	]
30	CTRL-^	62	>	94	^
31	n/a	63	?	95	— n/a

On the Apple screen, ASCII values 96 to 255 repeat the characters in column 2 first and then repeat the entire series again. A printer, however, will print some of them differently.

APPENDIX B
FILE TYPE USE, LISTED BY DOS COMMAND

DOS Command Uses	*Integer BASIC Program*	*Applesoft BASIC Program*	*Sequential Access Text*	*Random Access Text*	*Machine Language Binary*
Files:	***File***	***File***	***File***	***File***	***File***
APPEND			X		X
BLOAD					X
BRUN					X
BSAVE					X
CHAIN	X				
CLOSE			X	X	
DELETE	X	X	X	X	X
EXEC			X		
INIT	X	X			
LOAD	X	X			
LOCK	X	X	X	X	X
OPEN			X	X	
POSITION			X		
READ			X	X	
RENAME	X	X	X	X	X
RUN	X	X			
SAVE	X	X			
UNLOCK	X	X	X	X	X
VERIFY	X	X	X	X	X
WRITE			X	X	

Use these commands *only* in deferred execution mode:

APPEND, OPEN, POSITION, READ, WRITE

Copyright 1989, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

APPENDIX C
DOS COMMANDS (Execution Modes)

Deferred-Execution Mode

These commands should be used only in PRINT statements beginning with CTRL-D:

OPEN
APPEND
READ
WRITE
POSITION

Immediate-Execution Mode

These commands may be used in immediate-execution mode, and also from within a program using a PRINT statement beginning with CTRL-D:

CATALOG	BSAVE
SAVE	BLOAD
LOAD	BRUN
RUN	EXEC
DELETE	CLOSE
RENAME	CHAIN
LOCK and UNLOCK	PR#
MON and NOMON	IN#
MAXFILES	

Use INIT only in immediate-execution mode.

Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

APPENDIX D
DOS COMMANDS—CATEGORIES

Housekeeping Commands

INIT	RENAME	VERIFY
CATALOG	DELETE	MON
SAVE	LOCK	NOMON
LOAD	UNLOCK	MAXFILES
RUN		

Access Commands

FP	PR#	CHAIN
INT	IN#	

Sequential Text File Commands

OPEN	APPEND
CLOSE	POSITION
READ	EXEC
WRITE	

Random-Access Text File Commands

OPEN	READ
CLOSE	WRITE

Machine Language File Commands

BSAVE
BLOAD
BRUN

Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

APPENDIX E
FILE PARAMETERS

	Parameter	Letter	Range Minimum	Range Maximum
All Files	Slot	S	1	7
	Drive	D	1	2
	Volume	V	0*	254
Sequential Text Files	Byte	B	0	32767
	Relative Field	R	0	32767
	Absolute Field (EXEC)	R	0	32767
Random-Access Text Files	Record Length	L	1	32767
	Record Number	R	0	32767
Binary Files	Starting Address	A	0	65535
	Number of Bytes	L	1	32767

DOS Command	Quantity	Range Minimum	Range Maximum
PR# s	s	0	16†
IN# s	s	0	16†
MAXFILES n	n	1	16

*Minimum volume number INIT will actually *assign* to a diskette is 1.

†Maximum slot number built into the APPLE II is 7. In deferred-execution mode only, the SYNTAX ERROR message is given for s values from 8 through 16.

Note: The use of values outside the above ranges does not always cause the RANGE ERROR message. Any DOS command parameter or command quantity that is less than 0 or greater than 65535 will cause the SYNTAX ERROR message, *not* the RANGE ERROR message.

Copyright 1980, 1981, Apple Computer, Inc. Used by permission of Apple Computer, Inc., 20525 Mariani, Cupertino, CA 95014.

APPENDIX F
RESERVED WORDS IN APPLESOFT

&							
ABS	AND	ASC	AT	ATN			
CALL	CHR$	CLEAR	COLOR=	CONT	COS		
DATA	DEF	DEL	DIM	DRAW			
END	EXP						
FLASH	FN	FOR	FRE				
GET	GOSUB	GOTO	GR				
HCOLOR=	HGR	HGR2	HIMEM:	HLIN	HOME	HPLOT	HTAB
IF	IN#	INPUT	INT	INVERSE			
LEFT$	LEN	LET	LIST	LOAD	LOG	LOMEM:	
MID$							
NEW	NEXT	NORMAL	NOT	NOTRACE			
ON	ONERR	OR					
PDL	PEEK	PLOT	POKE	POP	POS	PRINT	PR#
READ	RECALL	REM	RESTORE	RESUME	RETURN	RIGHT$	RND
ROT=	RUN						
SAVE	SCALE=	SCRN(	SGN	SHLOAD	SIN	SPC(	SPEED=
SQR	STEP	STOP	STORE	STR$			
TAB(	TAN	TEXT	THEN	TO	TRACE		
USR							
VAL	VLIN	VTAB					
WAIT							
XPLOT	XDRAW						

Glossary

APPLE (1) An edible fruit. (2) The name of a well-known computer.

APPLESOFT A version of BASIC used by the APPLE computer.

ASCII An acronym that stands for "American Standard Code for Information Interchange." It is pronounced "ask-key." All characters, such as letters, numbers, control characters, etc., have a unique value.

BASIC Beginners' All-purpose Symbolic Instruction Code. A programming language used in some personal computers.

BIT Abbreviation for Binary digIT, the smallest unit of data recognized by a computer.

BOARD The circuit board. Holds the chips and electronic components of the computer.

BOOT An expression that means the process of inserting the disk, powering on the computer, and putting a program into memory.

BYTE A basic number of units in the computer's memory that makes a character recognizable to the computer, usually 8 or 16. An ASCII character is represented by one byte.

CATALOG A listing of the programs on a disk.

CHARACTER Any symbol, letter, number, space, etc.

CHIP Electronic circuit components embedded in a tiny silicon wafer in the computer.

COBOL Common Business Oriented Language. A programming language well suited to business applications.

CP/M An operating system used by some microcomputers.

CURSOR A blinking or solid marker to indicate where you are working on the screen.

DAISY-WHEEL PRINTER Letter-quality printer which uses a thimble or small print wheel to print fully formed characters.

DATA A representation of information to be used by the computer.

DATABASE A large collection of organized data required to perform a task.

DATA PROCESSING Operating on data usually done according to a set of rules, resulting in the solution to a problem.

DISK, DISKETTE Magnetic medium used to store data. Also called a *floppy*.

DISK DRIVE Equipment (hardware) used to record or play disks.

DOS Disk Operating System. Software that lets the user transfer programs and data between disk and computer.

DOT-MATRIX PRINTER A printer that forms characters from a two-dimensional array of dots. More dots produce a better image.

EDIT To correct, insert, delete or rearrange material.

ERROR MESSAGE Printed message that informs the user of a programming error. Can indicate type of error and location.

EXECUTE To perform, to carry out a program or command.

FORTRAN FORmula TRANslation. A high-level problem-oriented language used primarily by scientists and engineers.

HARD COPY Data printed in a permanent form (usually on paper) rather than shown temporarily on a computer screen.

HARD DISK A nonflexible disk that can store much more data than a floppy disk.

HARDWARE Physical equipment that makes up a computer system. This includes the monitor, central processing unit (CPU), printer, disk drive, etc.

INITIALIZE A DISK Transfer the disk operating system to the disk and clear any data.

INPUT The process of transferring data to the computer, as from the keyboard.

INTERFACE A piece of hardware connecting two pieces of equipment, making them compatible: an adapter.

KILOBYTE (KB) 1024 bytes, also referred to as 1K.

KEYBOARD An array of keys (much like a typewriter) used to enter data in a computer.

LOAD The process of transferring a program from a disk or cassette into the computer's memory.

LOGO A graphics-oriented programming language often used for children's programs.

MEGABYTE (MB) One million bytes.

MEMORY The part of the computer that stores data.

MENU A listing on the screen of programs available or of options available within a specific program.

MICROCOMPUTER A small computer, often referred to as a *personal computer*.

MODEM Modulator/Demodulator. A device that allows transfer of data between computers. The most common transmission device is the ordinary telephone.

MONITOR A television-like screen that displays output from the computer. Also called a VDT.

PASCAL A popular high-level programming language developed specifically for use with structured programming and top-down design.

PC Personal computer.

PEEK A way of examining or "peeking" directly into the contents of a particular memory location.

POKE A way of changing the contents of a particular memory location.

PRINTER A typewriter-like device that produces a copy or paper of data stored in the computer.

RAM Random Access Memory. Computer memory that can be read or changed (written to) during normal operation. Turning the computer off destroys the contents of RAM.

ROM Read-only Memory. Computer memory where information is stored permanently during manufacturing. The computer can use it, but cannot change it.

SAVE To store a program or data on disk for later retrieval. Programs SAVEd to a disk are not lost when the power is turned off.

SECTORS A specific area of a disk track.

SOFTWARE Computer programs.

TERMINAL A device used to enter data into a computer and record the output. Usually consists of a keyboard and a video display or printer.

TEXT Written information.

TEXT FILES A file that stores text.

TRACK A specific recording path on a disk.

VDT Video display terminal. See Monitor.

WORD PROCESSING Manipulating text electronically, performing such functions as paragraphing, inserting, deleting, or rearranging information, formatting it, paging, etc.

Bibliography

The Applesoft Tutorial. Cupertino, CA: Apple Computer Inc., 1981.

Apple II Reference Manual. Cupertino, CA: Apple Computer Inc., 1981.

BASIC Programming Reference Manual. Cupertino, CA: Apple Computer Inc., 1981.

Finkel, LeRoy and Jerald R. Brown *Apple Basic: Data File Programming* New York, NY: John Wiley & Sons, Inc., 1982.

Fudge, Don, "Fudge It," *inCider* (May 1983), pp. 158–159.

Luebbert, William F., *What's Where in the Apple, A Complete Guide to the Apple Computer.* Chelmsford, MS: Micro Ink, 1981.

Public Domain Software Library

Index

A

Address, 26
Alphabetizing:
  routines, 126–28
  text files (*see* Random access text file; Sequential text file)
Ampersand:
  kill, 126
  uses, 31–32, 125
  vector or hook, 10
And hook (*see* Ampersand)
Annunciator outputs, 30
APPEND, 186–88
Arrays, 136
ASCII:
  chart, 236
  definition, 102, 242
  table generator, 132
  uses, 102–4, 144–45, 174–76
Averaging numbers, 130

B

Back slash, 90
Buffer (*see* MAXFILES)
Byte:
  control:
    random access text file, 226–30
  sequential text file, 201–4
  definition, 171, 242
  overhead, 171, 173, 206–7

C

CALL:
  chart, 32–33
  definition, 28
CATALOG:
  definition, 162–63, 242
  lister, 125
  utility, 126
Centering routines, 85–89
CLOSE (*see* Disk commands)
Column alignment, 128–30
Concatenation, 109
Control characters, 102, 109–10, 142–43
Cursor:
  control, 38–46, 109–11
  dash, 126
  inverted V, 75
  invisible, 76
  locations, 29, 31

D

Data:
  definition, 213
  file, 162
  set, 183, 205–7
Debug (*see* Error)
Deferred execution mode, 113, 115, 174, 238
Disk:
  commands, 174–76
  format, 171–74
  operating system (*see* DOS)
Disk commands, 174–76
DOS:
  commands:
    categories, 239
    execution modes, 238
    file type use, 237
  definition, 243
  error codes, 118–23
  limitations, 122, 139–40
  related errors, 115–21, 123

E

Editing Pokes, 30, 138–39
Error:
  codes, 118–24
    DOS, 118–22
    BASIC, 121–24
  definition, 116–17
  DOS related, 115–21, 123
  ONERR, 31, 116–22
  PEEK location, 29, 116–17
  programming related, 107–14, 121–24
  trapping, 99–124
Execution mode (*see* Deferred execution mode; Immediate execution mode)
EXTRA IGNORED?, 94–95, 108, 183

F

FID program, 192
Field, 183
File:
  buffer, 169
  parameters, 240
  text (*see* Text file)
  type use, 237
Fill routine, 52–53
FLASH, 74–75
Frames, creation:
  hello, 83
  inverse box, 80–83
  star, 83–84
Functions (*see* String functions)

G

Games, 145–52
Glossary, 242–44
Graphic pokes, 30, 32

H

HTAB:
  definition, 38
  grid, 39

I

Immediate execution mode, 113, 115, 174, 238
INPUT:
  definition, 243
  error proof, 96
  error trapping, 100–112
  format, 220–21
  interrupts, 108
  inverse field, 92–94
  limiting, 45, 107–8
  techniques, 89–91, 143–45
INVERSE, 73–74, 80–83

J

Joystick:
  connected, 132
  control, 29
  tester, 132–33

K

Keyboard, read the, 95–96
Keywords, 223–26

M

Margins:
  bottom, 29, 37
  chart, 36
  left, 29, 35
  manipulation, 56–69
  top, 29, 37
  width, 29, 35, 138
Mathematical routines:
  averaging, 130
  mortgage, 131
  probability, 131–32
MAXFILES, 168–71
Menu, 83, 90, 97, 103
Modular programming, 163–66
MON, 115–16, 182
Monophonic, 1
Music:
  examples, 9–14
  monophonic, 1
  notes and values, 7
  poke for music, 10–11
  polyphonic, 1

N

NOMON, 115–16
Null character, 103, 110, 195

O

OPEN (*see* Disk commands)

P

Padding (*see* Random access text file)
Paddles (*see* Joystick)

PEEK:
chart, 29–30
definition, 27, 244
Pointer, internal, 186–87
POKE:
chart, 30–32
definition, 27, 244
protection, 142 (*see also* Security procedures)
Polyphonic, 1
POSITION, 197–201
Printer output format, 138–39
Public Domain Software (*see* Software)

Q

Quotation marks, 85

R

Random access text file:
adding to, 215–17
alphabetizing, 223
byte control (*see* Byte)
changes, making, 218–21
copying, 221–23
creating a file, 207–13
deleting, 221
fixed length, 173, 206, 208, 213
limitations, 173
padding, 228–30
record, 173, 205–7
retrieving data, 213–15
searching, 223–26
structure, 173–74
Random numbers, 153–54
READ (*see* Disk commands)
Reserved words (listing), 241
Reset key, 116, 121–22
RTG (*see* Tone generator)

S

Scrolling effects:
horizontal, 71–73, 78, 81–82, 84–85
vertical, 69–70, 77–79, 97
Sector, 171–72, 244
Security procedures, 31–32, 46–47, 116, 121–22, 141–45
Sequential text file:
adding to, 186–88
alphabetizing, 195–97
byte control (*see* Byte)
changes, making, 188–91
copying, 192–94
creating a file, 178–83
deleting, 191
examples, 177
limitations, 173
position, 197–201
retrieving data, 183–86
structure, 172–73
Software:
definition, 244
free, 231–34
public domain, 233–34
Sort routines (*see* Alphabetizing)
Sound (*see also* Music)
clicking the speaker, 2, 29–30
effects, 15, 20–25, 80
ring bell, 33, 140–41
tone generator, 2, 145–52
SPC, 41
SPEED, 31, 55, 68, 113–14
Spin:
input, 89–90
menu, 90–91
writing, 91–92
String functions:
ASC, 102–3
CHR$, 103–4, 174
LEFT$, 104–5
LEN, 100

MID$, 105–6
RIGHT$, 105
STR$, 101–2
VAL, 100–101
Strobe, 90

T

TEXT, 61
Text file:
definition, 162, 244
types (*see* Random access text file; Sequential text file)
Text manipulation:
centering, 85–89
quotation marks, 85
underlining, 76
Text window, 34 (*see also* Margins)
Tone generator, 2–6, 8
Tone routine (*see* Tone generator)
Tone tester, 8
TRACE, 112–13
Tracks, 171–72
Twirling cursor (*see* Spin)

U

Underlining text, 76

V

Variables:
names, 166
string 167–68 (*see also* String functions)
VTAB:
definition, 38
grid, 39

W

WRITE (*see* Disk commands)

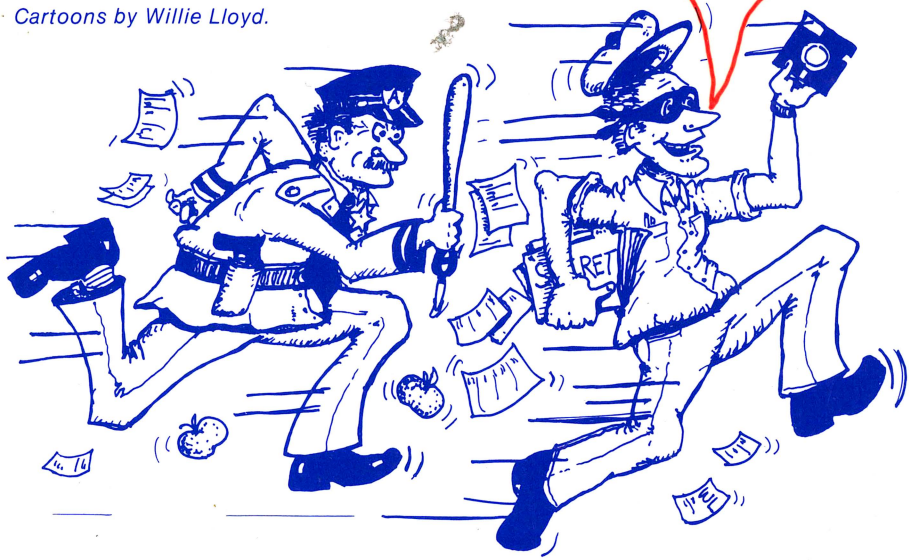
Disclosed within the pages of this book are many programming secrets never before published.

You'll learn how to:

- play "Name That Tune" with the computer;
- make screen clears look like dramatic scene changes on television;
- create moving flashing banners and billboards;
- recover files you thought were lost forever;
- learn how to use POKE, PEEK, and CALL;
- make the computer do what you want by controlling the cursor and screen display;
- protect your programs from prying eyes and write secret messages;
- trap and prevent programming and data entry errors;
- store information efficiently and retrieve it quickly through the use of text files;

—plus much more!!!

Cartoons by Willie Lloyd.



Never before have so many secrets been revealed at one time!!!

Apple Programming Secrets "THEY" didn't want you to know

is just too good to keep quiet about.

Wayne Dyrness

PRENTICE-HALL
Englewood Cliffs, N.J. 07632

ISBN 0-13-039215-4

Dyrness

Apple Programming Secrets "THEY" didn't want you to know

PRENTICE-HALL